IBM DB2 Universal Database

Call Level Interface Guide and Reference

Version 6

SC09-2843-00

IBM DB2 Universal Database

Call Level Interface Guide and Reference

Version 6

SC09-2843-00

Before using this information and the product it supports, be sure to read the general information under “Appendix M. Notices” on page 883.

Contents

About This Book

This book provides the information necessary to write applications using DB2 Call Level Interface to access the IBM DB2 family of servers. This book should also be used as a supplement when writing applications (using an ODBC Software Development Kit) that access the IBM DB2 family of servers using ODBC.

References in this book to 'DB2', with or without a product version number, should be understood to mean the 'DB2 Universal Database' product. Reference to DB2 on other platforms use the specific product name (such as DB2 for OS/390 or DB2 for AS/400).

Who Should Use this book

DB2 application programmers with a knowledge of SQL and the 'C' or 'C++' programming language.

ODBC application programmers with a knowledge of SQL and the 'C' or 'C++' programming language.

How this Book is Structured

This book is divided into the following chapters:

- "Chapter 1. Introduction to CLI" on page 1, introduces DB2 CLI and discusses the background of the interface and its relation to embedded SQL.
- "Chapter 2. Writing a DB2 CLI Application" on page 9, provides an overview of a typical DB2 CLI application. This chapter discusses the basic tasks or steps within a simple DB2 CLI application. General concepts are introduced as well as the basic functions and the interaction between them.
- "Chapter 3. Using Advanced Features" on page 41, provides an overview of more advanced tasks and the functions used to perform them.
- "Chapter 4. Configuring CLI/ODBC and Running Sample Applications" on page 149, contains information for setting up the necessary environment to compile and run DB2 CLI applications. Sample applications are provided in order to verify your environment, and to provide templates to help you develop your own applications. A listing of the CLI/ODBC configuration keywords and their meanings is also included in this chapter.

- "Chapter 5. DB2 CLI Functions" on page 209, is a reference for the functions that make up DB2 CLI.
- Appendixes:
  - "Appendix A. Programming Hints and Tips" on page 757, provides some common hints and tips for improving performance and/or portability of DB2 CLI applications.
  - "Appendix B. Migrating Applications" on page 767, summarizes what has changed since the last release, and any incompatibilities and necessary steps required for migrating existing applications.
  - "Appendix C. DB2 CLI and ODBC" on page 779, discusses the differences between ODBC and DB2 CLI.
  - "Appendix D. Extended Scalar Functions" on page 785, describes the scalar functions that can be accessed as DB2 functions, or using ODBC vendor escape clauses.
  - "Appendix E. SQLSTATE Cross Reference" on page 799, contains an SQLSTATE table that lists the functions that may generate each SQLSTATE. (Each function description in "Chapter 5. DB2 CLI Functions" on page 209 lists the possible SQLSTATEs for each function.)
  - "Appendix F. Data Conversion" on page 817, contains information about SQL and C data types, and conversion between them.
  - "Appendix G. Catalog Views for Stored Procedures" on page 837, contains a description of the catalog tables used to store information about stored procedures and their attributes.
  - "Appendix H. Pseudo Catalog Table for Stored Procedure Registration" on page 839, describes how to create and maintain the DB2CLI.PROCEDURES pseduo-catalog table.
  - "Appendix I. Supported SQL Statements" on page 843, contains a list of the SQL statements supported by DB2 Universal Database, and the subset of SQL supported by DB2 CLI.
  - "Appendix J. CLI Sample Code" on page 847, lists the complete source for some extensive examples.
  - "Appendix K. Using the DB2 CLI/ODBC Trace Facility" on page 857, contains information about the trace file format, tracing asynchronous applications, and related topics.

Chapter 1. Introduction to CLI

DB2 Call Level Interface (DB2 CLI) is IBM's callable SQL interface to the DB2 family of database servers. It is a 'C' and 'C++' application programming interface for relational database access, and it uses function calls to pass dynamic SQL statements as function arguments. It is an alternative to embedded dynamic SQL, but unlike embedded SQL, it does not require host variables or a precompiler.

DB2 CLI is based on the Microsoft** Open Database Connectivity** (ODBC) specification, and the International Standard for SQL/CLI. These specifications were chosen as the basis for the DB2 Call Level Interface in an effort to follow industry standards and to provide a shorter learning curve for those application programmers already familiar with either of these database interfaces. In addition, some DB2 specific extensions have been added to help the application programmer specifically exploit DB2 features.

The DB2 CLI driver also acts as an ODBC driver when loaded by an ODBC driver manager. It conforms to level 2 of ODBC 2.0, and level 1 of ODBC 3.0. In addition, it also conforms to various ODBC 3.0 level 2 interface conformance items (202, 203, 205, 207, 209, and 211). Information regarding ODBC support and level 2 interface conformance items is provided in "Appendix C. DB2 CLI and ODBC" on page 779.

DB2 CLI Background Information

To understand DB2 CLI or any callable SQL interface, it is helpful to understand what it is based on, and to compare it with existing interfaces.

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface referred to as the *X/Open Call Level Interface*. The goal of this interface is to increase the portability of applications by enabling them to become independent of any one database vendor's programming interface. Most of the X/Open Call Level Interface specification has been accepted as part of the ISO Call Level Interface International Standard (ISO/IEC 9075-3:1995 SQL/CLI).

Microsoft developed a callable SQL interface called Open Database Connectivity (ODBC) for Microsoft operating systems based on a preliminary draft of X/Open CLI.

Version 3 of ODBC conforms to almost all of ISO SQL/CLI. ODBC 3.0 does contain considerable functionality that is not part of the International Standard; much of this is being added to the the next draft of the standard.

The ODBC specification also includes an operating environment where database specific ODBC Drivers are dynamically loaded at run time by a driver manager based on the data source (database name) provided on the connect request. The application is linked directly to a single driver manager library rather than to each DBMS's library. The driver manager mediates the application's function calls at run time and ensures they are directed to the appropriate DBMS specific ODBC driver. Since the ODBC driver manager only knows about the ODBC-specific functions, DBMS-specific functions cannot be accessed in an ODBC environment. DBMS-specific dynamic SQL statements are supported via a mechanism called an escape clause which is described in "Using Vendor Escape Clauses" on page 144.

ODBC is not limited to Microsoft operating systems; other implementations are available on various platforms.

The DB2 CLI load library can be loaded as an ODBC driver by an ODBC driver manager. For ODBC application development, you must obtain an ODBC Software Development Kit (from Microsoft for Microsoft platforms, and from other vendors for non-Microsoft platforms.) When developing ODBC applications that may connect to DB2 servers, use this book (for information on DB2 specific extensions and diagnostic information), in conjunction with the *ODBC 3.0 Programmer's Reference and SDK Guide*.

Applications written directly to DB2 CLI link directly to the DB2 CLI load library. DB2 CLI includes support for many ODBC and ISO SQL/CLI functions, as well as DB2 specific functions. For a list of supported functions, refer to "DB2 CLI Function Summary" on page 211.

For more information on the relationship between DB2 CLI and ODBC, refer to "Appendix C. DB2 CLI and ODBC" on page 779.

The following DB2 features are available to both ODBC and DB2 CLI applications:

- The double byte (graphic) data types
- Stored Procedures
- Distributed Unit of Work (DUOW), two phase commit
- Compound SQL
- User Defined Types (UDT)
- User Defined Functions (UDF)

DB2 CLI also contains extensions to access DB2 features that can not be accessed by ODBC applications:

- Support of Large Objects (LOBs), and LOB locators
- SQLCA access for detailed DB2 specific diagnostic information

Differences Between DB2 CLI and Embedded 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 DRDA databases. They just connect to the appropriate database at run time.

Comparing Embedded SQL and DB2 CLI

DB2 CLI and embedded SQL also differ in the following ways:

- 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 issued via the `SQLEndTran()` function call rather than by passing it as an SQL statement.
- DB2 CLI manages statement related information on behalf of the application, and provides a *statement handle* to refer to it as an abstract object. 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 all 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 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). Refer to "SQLState Cross Reference table" on page 799 for a cross reference of all DB2 CLI SQLSTATEs.
- DB2 CLI supports scrollable cursors. With scrollable cursors, you can scroll through a static 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.

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.

Table 215 on page 843 lists each SQL statement, and indicates whether or not it can be executed using DB2 CLI. The table also indicates if the command line processor can be used to execute the statement interactively, (useful for prototyping SQL statements).

Each DBMS may have additional statements that you can dynamically prepare. In this case, DB2 CLI passes the statements to the DBMS. There is one exception: the COMMIT and ROLLBACK statement can be dynamically prepared by some DBMSs but are not passed. In this case, use the `SQLEndTran()` function to specify either the COMMIT or ROLLBACK statement.

Advantages of Using DB2 CLI

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 emerging 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 for MVS/ESA 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 server that is accessible from a DataJoiner Version 2 server.

- 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 read-only 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.
- 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.

Deciding on Embedded SQL or DB2 CLI

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.

For more information on using static SQL in CLI applications, refer to the Web page at:

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

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. This additional step (and network-traffic), however, will not be required if the DB2 CLI application makes use of deferred prepare.

It is important to note that static SQL will not always have better performance than dynamic SQL. Dynamic SQL can make use of changes to the database, such as new indexes, and can use current database statistics to choose the optimal access plan. In addition, precompilation of statements can be avoided if they are cached.

- Encapsulation and Security

  In static SQL, the authorizations to 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. For more information, refer to “Using Stored Procedures” on page 125.

For more information on using static SQL in CLI applications, refer to the Web page at:

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

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. Refer to “Mixing Embedded SQL and DB2 CLI” on page 136.

Ultimately, the decision on when to use each interface, will be based on individual preferences and previous experience rather than on any one factor.

Supported Environments

DB2 CLI run time support is provided by both the DB2 server and client. Refer to “Summary of Changes” on page 767 for information about support in previous versions.

DB2 CLI development support is included with the Software Developer’s Kit. The support consists of the necessary header files, link libraries and documentation required to develop both embedded and DB2 CLI applications for a particular operating environment. For example, DB2 SDK for OS/2 allows you to write applications that run under OS/2, with a DB2 client these applications can access data on a DB2 for OS/2, DB2 for AIX, or other DB2 Universal Database servers. With DB2 Connect, the applications can access DB2 Universal Database for AS/400, DB2 for OS/390, DB2 for VSE & VM servers or any other IBM or non-IBM DRDA server.

For detailed information on the DB2 Software Developer’s Kit, and for using supported compilers to build DB2 CLI applications, see the *Application Building Guide*.

Other Information Sources

When writing DB2 CLI applications, you may need to reference information for the database servers that are being accessed, in order to understand any connectivity issues, environment issues, SQL language support issues, and other server-specific information. For DB2 Universal Database versions, refer to *SQL Reference*, the *Application Development Guide*, and the *Administrative API Reference*. If you are writing applications that will access other DB2 server products, refer to the *SQL Reference* which contains information that is common to all products, including any differences.

Chapter 2. Writing a DB2 CLI Application

This section introduces a conceptual view of a typical DB2 CLI application.

A DB2 CLI application can be broken down into a set of tasks. Some of these tasks are organized into discrete steps, while others may apply throughout the application. Each task is carried out by calling one or more DB2 CLI functions.

Tasks described in this section are basic tasks that apply to all applications. More advanced tasks, such as using array insert or using large object support, are discussed in “Chapter 3. Using Advanced Features” on page 41.

The functions are used in examples to illustrate their use in DB2 CLI applications. Refer to “Chapter 5. DB2 CLI Functions” on page 209 for complete descriptions and usage information for each of the functions.

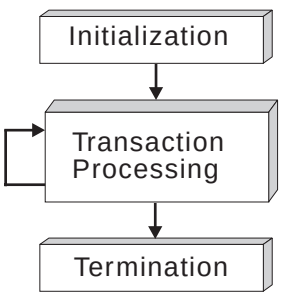


Figure 1. Conceptual View of a DB2 CLI Application

Every DB2 CLI application contains the three main tasks shown in Figure 1 on page 9.

Initialization
: This task allocates and initializes some resources in preparation for the main *Transaction Processing* task. Refer to "Initialization and Termination" for details.

Transaction Processing
: This is the main task of the application. SQL statements are passed to DB2 CLI to query and modify the data. Refer to "Transaction Processing" on page 15 for details.

Termination
: This task frees allocated resources. The resources generally consist of data areas identified by unique handles. Refer to "Initialization and Termination" for details.

As well as the three tasks listed above, there are general tasks, such as handling diagnostic messages, which occur throughout an application.

Initialization and Termination

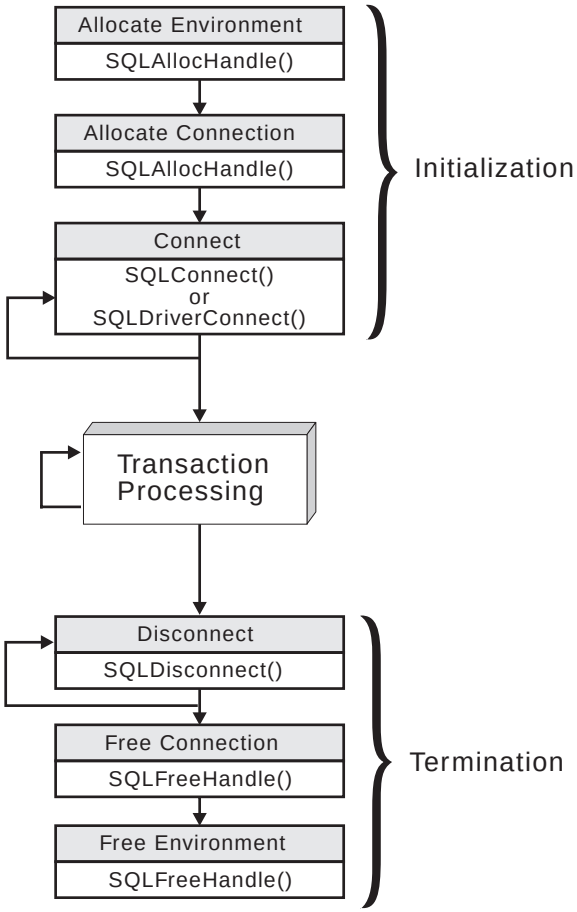


Figure 2. Conceptual View of Initialization and Termination Tasks

Figure 2 shows the function call sequences for both the initialization and termination tasks. The transaction processing task in the middle of the diagram is shown in Figure 3 on page 16.

Handles

The initialization task consists of the allocation and initialization of environment and connection handles (which are later freed in the termination task). An application then passes the appropriate handle when it calls other DB2 CLI functions. A handle is a variable that refers to a data object controlled by DB2 CLI. Using handles relieves the application from having to allocate and manage global variables or data structures, such as the SQLDA or SQLCA, used in IBM's embedded SQL interfaces.

The SQLAllocHandle() function is called with a handle type and parent handle arguments to create environment, connection, statement, or descriptor handles. The function SQLFreeHandle() is used to free the resources allocated to a handle.

There are four types of handles:

Environment Handle
The environment handle refers to the data object that contains information regarding the global state of the application, such as attributes and connections. An environment handle must be allocated before a connection handle can be allocated.

Connection Handle
A connection handle refers to a data object that contains information associated with a connection to a particular data source (database). This includes connection attributes, general status information, transaction status, and diagnostic information.

An application can be connected to several servers at the same time, and can establish several distinct connections to the same server. An application requires a connection handle for each concurrent connection to a database server. For information on multiple connections, refer to "Connecting to One or More Data Sources".

Call SQLGetInfo() to determine if a user imposed limit on the number of connector handles has been set.

Statement Handle(s)
Statement handles are discussed in the next section, "Transaction Processing" on page 15.

Descriptor Handle(s)
A descriptor handle refers to a data object that contains information about:

- columns in a result set
- dynamic parameters in an SQL statement

Descriptors and descriptor handles are discussed in the section "Using Descriptors" on page 97.

Connecting to One or More Data Sources

In order to connect concurrently to one or more data sources (or multiple concurrent connections to the same data source), an application calls SQLAllocHandle(), with a *HandleType* of SQL_HANDLE_DBC, once for each connection. The subsequent connection handle is used with SQLConnect() to request a database connection and with SQLAllocHandle(), with a *HandleType*

of SQL_HANDLE_STMT, to allocate statement handles for use within that connection. There is also an extended connect function, SQLDriverConnect(), which allows for additional connect options, and the ability to directly open a connection dialog box in environments that support a Graphical User Interface. The function SQLBrowseConnect() can be used to discover all of the attributes and attribute values required to connect to a data source.

The use of connection handles ensures that multi-threaded applications that utilize one connection per thread are thread-safe since separate data structures are allocated and maintained by DB2 CLI for each connection.

Unlike the distributed unit of work connections described in "Multisite Updates (Two Phase Commit)" on page 50, there is no coordination between the statements that are executed on different connections.

Initialization and Connection Example

```
/* From CLI sample basiccon.c */
/* ... */
#include <stdio.h>
#include <stdlib.h>
#include <sqlcli1.h>

/* ... */

SQLRETURN
prompted_connect( SQLHANDLE henv,
                  SQLHANDLE * hdbc);

#define MAX_UID_LENGTH   18
#define MAX_PWD_LENGTH   30
#define MAX_CONNECTIONS  2

#define MAX_CONNECTIONS 2

/* extern SQLCHAR server[SQL_MAX_DSN_LENGTH + 1] ;
extern SQLCHAR uid[MAX_UID_LENGTH + 1] ;
extern SQLCHAR pwd[MAX_PWD_LENGTH + 1] ;
*/

int main( ) {

    SQLHANDLE henv;
    SQLHANDLE hdbc[MAX_CONNECTIONS] ;

/* ... */

    /* allocate an environment handle */
    SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;

    /* Connect to first data source */
    prompted_connect( henv, &hdbc[0] ) ;
```

```
    /* Connect to second data source */
    prompted_connect( henv, &hdbc[1] ) ;

    /*********   Start Processing Step  ************************/
    /* allocate statement handle, execute statement, etc.       */
    /*********   End Processing Step  *************************/

    printf( "\nDisconnecting .....\n" ) ;
    SQLDisconnect( hdbc[0] ) ;  /* disconnect first connection */
    SQLDisconnect( hdbc[1] ) ;  /* disconnect second connection */

    /* free first connection handle */
    SQLFreeHandle( SQL_HANDLE_DBC, hdbc[0] ) ;

    /* free second connection handle */
    SQLFreeHandle( SQL_HANDLE_DBC, hdbc[1] ) ;

    /* free environment handle */
    SQLFreeHandle( SQL_HANDLE_ENV, henv ) ;

    return ( SQL_SUCCESS ) ;

}

/* prompted_connect - prompt for connect options and connect */
SQLRETURN prompted_connect( SQLHANDLE henv,
                            SQLHANDLE * hdbc
                          ) {

    SQLCHAR server[SQL_MAX_DSN_LENGTH + 1] ;
    SQLCHAR uid[MAX_UID_LENGTH + 1] ;
    SQLCHAR pwd[MAX_PWD_LENGTH + 1] ;

    /* allocate a connection handle     */
    if ( SQLAllocHandle( SQL_HANDLE_DBC,
                         henv,
                         hdbc
                       ) != SQL_SUCCESS ) {
        printf( ">---ERROR while allocating a connection handle-----\n" ) ;
        return( SQL_ERROR ) ;
    }

    /* Set AUTOCOMMIT OFF */
    if ( SQLSetConnectAttr( * hdbc,
                            SQL_ATTR_AUTOCOMMIT,
                            ( void * ) SQL_AUTOCOMMIT_OFF, SQL_NTS
                          ) != SQL_SUCCESS ) {
       printf( ">---ERROR while setting AUTOCOMMIT OFF ------------\n" ) ;
       return( SQL_ERROR ) ;
    }

    printf( ">Enter Server Name:\n" ) ;
    gets( ( char * ) server ) ;
    printf( ">Enter User Name:\n" ) ;
```

```
    gets( ( char * ) uid ) ;
    printf( ">Enter Password:\n" ) ;
    gets( ( char * ) pwd ) ;

    if ( SQLConnect( * hdbc,
                     server, SQL_NTS,
                     uid,    SQL_NTS,
                     pwd,    SQL_NTS
                   ) != SQL_SUCCESS ) {
        printf( ">--- ERROR while connecting to %s -------------\n",
                server
              ) ;

        SQLDisconnect( * hdbc ) ;
        SQLFreeHandle( SQL_HANDLE_DBC, * hdbc ) ;
        return( SQL_ERROR ) ;
    }
    else              /* Print Connection Information */
        printf( "Successful Connect to %s\n", server ) ;

    return( SQL_SUCCESS ) ;

}
```

Transaction Processing

The following figure shows the typical order of function calls in a DB2 CLI application. Not all functions or possible paths are shown.

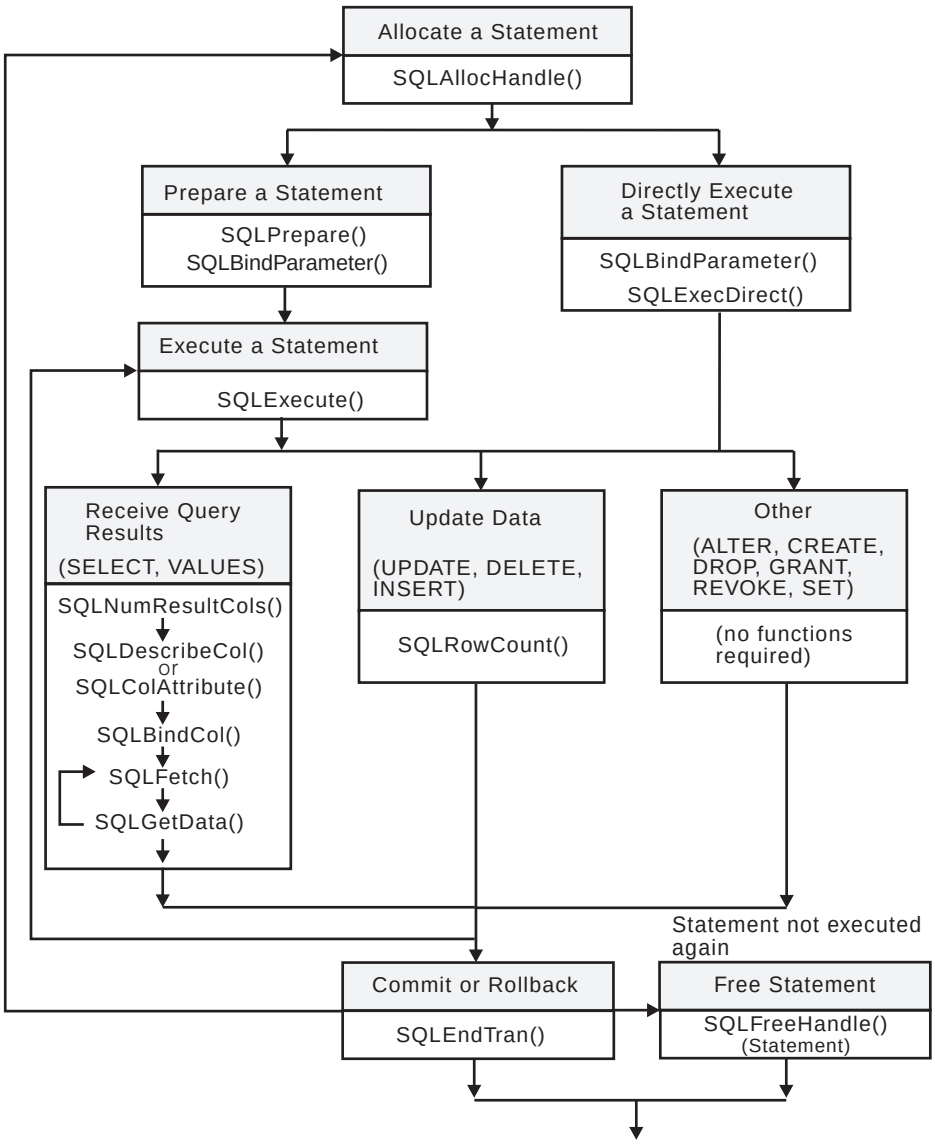


Figure 3. Transaction Processing

Figure 3 shows the steps and the DB2 CLI functions in the transaction processing task. This task contains five steps:

- Allocating statement handle(s)
- Preparation and execution of SQL statements
- Processing results
- Commit or Rollback

- Optionally, Freeing statement handle(s) if the statement is unlikely to be executed again.

Allocating Statement Handle(s)

`SQLAllocHandle()` is called with a *HandleType* of SQL_HANDLE_STMT to allocate a statement handle. A statement handle refers to the data object that is used to track the execution of a single SQL statement. This includes information such as statement attributes, SQL statement text, dynamic parameters, cursor information, bindings for dynamic arguments and columns, result values and status information (these are discussed later). Each statement handle is associated with a connection handle.

A statement handle must be allocated before a statement can be executed.

The maximum number of statement handles that may be allocated at any one time is limited by overall system resources (usually stack size). The maximum number of statement handles that may actually be used, however, is defined by DB2 CLI: 5500. An HY014 SQLSTATE will be returned on the call to `SQLPrepare()` or `SQLExecDirect()` if the application exceeds these limits.

Preparation and Execution

Once a statement handle has been allocated, there are two methods of specifying and executing SQL statements:

1. Prepare then execute
   a. Call `SQLPrepare()` with an SQL statement as an argument.
   b. Call `SQLBindParameter()` if the SQL statement contains *parameter markers*.
   c. Call `SQLExecute()`
2. Execute direct
   a. Call `SQLBindParameter()` if the SQL statement contains *parameter markers*.
   b. Call `SQLExecDirect()` with an SQL statement as an argument.

The first method splits the preparation of the statement from the execution. This method is used when:

- The statement will be executed repeatedly (usually with different parameter values). This avoids having to prepare the same statement more than once. The subsequent executions make use of the access plans already generated by the prepare.
- The application requires information about the columns in the result set, prior to statement execution.

The second method combines the prepare step and the execute step into one. This method is used when:

- The statement will be executed only once. This avoids having to call two functions to execute the statement.
- The application does not require information about the columns in the result set, before the statement is executed.

Note: `SQLGetTypeInfo()` and the schema (catalog) functions discussed in "Chapter 3. Using Advanced Features" on page 41, execute their own query statements, and generate a result set. Calling a schema function is equivalent to executing a query statement, the result set is then processed as if a query statement had been executed.

DB2 Universal Database version 5 or later has a *global dynamic statement cache* stored on the server. This cache is used to store the most popular access plans for prepared SQL statements. Before each statement is prepared, the server searches this cache to see if an access plan has already been created for this exact SQL statement (by this application or any other application or client). If so, the server does not need to generate a new access plan, but will use the one in the cache instead. There is now no need for the application to cache connections at the client unless connecting to a server that does not have a global dynamic statement cache (such as DB2 Common Server v2). For information on caching connections at the client see "Caching Statement Handles on the Client" on page 770 in the Migration section.

Binding Parameters in SQL Statements: Both of the execution methods described above allow the use of parameter markers in place of an *expression* (or host variable in embedded SQL) in an SQL statement.

Parameter markers are represented by the '?' character and indicate the position in the SQL statement where the contents of application variables are to be substituted when the statement is executed. The parameter markers are referenced sequentially, from left to right, starting at 1. `SQLNumParams()` can be used to determine the number of parameters in a statement.

When an application variable is associated with a parameter marker it is *bound* to the parameter marker. The application must bind an application variable to each parameter marker in the SQL statement before it executes that statement. Binding is carried out by calling the `SQLBindParameter()` function with a number of arguments to indicate, the numerical position of the parameter, the SQL type of the parameter, the data type of the variable, a pointer to the application variable, and length of the variable.

The bound application variable and its associated length are called *deferred* input arguments since only the pointers are passed when the parameter is

bound; no data is read from the variable until the statement is executed. Deferred arguments allow the application to modify the contents of the bound parameter variables, and repeat the execution of the statement with the new values.

Information for each parameter remains in effect until overridden, or until the application unbinds the parameter or drops the statement handle. If the application executes the SQL statement repeatedly without changing the parameter binding, then DB2 CLI uses the same pointers to locate the data on each execution. The application can also change the parameter binding to a different set of deferred variables. The application must not de-allocate or discard variables used for deferred input fields between the time it binds the fields to parameter markers and the time DB2 CLI accesses them at execution time.

It is possible to bind the parameters to a variable of a different type from that required by the SQL statement. The application must indicate the C data type of the source, and the SQL type of the parameter marker, and DB2 CLI will convert the contents of the variable to match the SQL data type specified. For example, the SQL statement may require an integer value, but your application has a string representation of an integer. The string can be bound to the parameter, and DB2 CLI will convert the string to the corresponding integer value when you execute the statement.

By default, DB2 CLI does not verify the type of the parameter marker. If the application indicates an incorrect type for the parameter marker, it could cause either an extra conversion by the DBMS, or an error. Refer to “Data Types and Data Conversion” on page 28 for more information about data conversion.

Information about the parameter markers can be accessed using descriptors. If you enable automatic population of the implementation parameter descriptor (IPD) then information about the parameter markers will be collected. The statement attribute SQL_ATTR_ENABLE_AUTO_IPD must be set to SQL_TRUE for this to work. See “Using Descriptors” on page 97 for more information.

If the parameter marker is part of a predicate on a query and is associated with a User Defined Type, then the parameter marker must be cast to the built-in type in the predicate portion of the statement; otherwise, an error will occur. For an example, refer to “User Defined Types in Predicates” on page 776.

The global dynamic statement cache was introduced in an earlier section. The access plan will only be shared between statements if they are exactly the

same. For SQL statements with parameter markers, the specific values that are bound to the parameters do not have to be the same, only the SQL statement itself.

For information on more advanced methods for binding application storage to parameter markers, refer to:

- "Using Arrays to Input Parameter Values" on page 83
- "Sending/Retrieving Long Data in Pieces" on page 80
- "Parameter Binding Offsets" on page 87

Processing Results

The next step after the statement has been executed depends on the type of SQL statement.

Processing Query (SELECT, VALUES) Statements: If the statement is a query statement, the following steps are generally needed in order to retrieve each row of the result set:

1. Establish (describe) the structure of the result set, number of columns, column types and lengths
2. (Optionally) bind application variables to columns in order to receive the data
3. Repeatedly fetch the next row of data, and receive it into the bound application variables
4. (Optionally) retrieve columns that were not previously bound, by calling `SQLGetData()` after each successful fetch.

- Each of the above steps requires some diagnostic checks.
- "Chapter 3. Using Advanced Features" on page 41 discusses advanced techniques of using `SQLFetchScroll()` to fetch multiple rows at a time.
- DB2 CLI also supports scrollable read-only cursors; see "Scrollable Cursors" on page 68 for more information.

Step 1 The first step requires analyzing the executed or prepared statement. The application will need to query the number of columns, the type of each column, and perhaps the names of each column in the result set. This information can be obtained by calling `SQLNumResultCols()` and `SQLDescribeCol()` (or `SQLColAttributes()`) after preparing or after executing the statement.

Step 2 The second step allows the application to retrieve column data directly into an application variable on the next call to `SQLFetch()`. For each column to be retrieved, the application calls `SQLBindCol()` to bind an application variable to a column in the result set. The application may use the information obtained from Step 1 to

determine the C data type of the application variable and to allocate the maximum storage the column value could occupy. Similar to variables bound to parameter markers using SQLBindParameter(), columns are bound to deferred arguments. This time the variables are deferred output arguments, as data is written to these storage locations when SQLFetch() is called.

If the application does not bind any columns, as in the case when it needs to retrieve columns of long data in pieces, it can use SQLGetData(). Both the SQLBindCol() and SQLGetData() techniques can be combined if some columns are bound and some are unbound. The application must not de-allocate or discard variables used for deferred output fields between the time it binds them to columns of the result set and the time DB2 CLI writes the data to these fields.

Step 3 The third step is to call SQLFetch() to fetch the first or next row of the result set. If any columns have been bound, the application variable will be updated. SQLFetchScroll() can also be used for added flexibility when moving through the result set, refer to "Scrollable Cursors" on page 68 for more information. SQLFetchScroll() can also be used by the application to fetch multiple rows of the result set into an array. Refer to "Retrieving a Result Set into an Array" on page 90 for more information.

If data conversion was indicated by the data types specified on the call to SQLBindCol(), the conversion will occur when SQLFetch() is called. Refer to "Data Types and Data Conversion" on page 28 for an explanation.

Step 4 (Optional)
The last (optional) step, is to call SQLGetData() to retrieve any unbound columns. All columns can be retrieved this way, provided they were not bound. SQLGetData() can also be called repeatedly to retrieve large columns in smaller pieces, which cannot be done with bound columns.

Data conversion can also be indicated here, as in SQLBindCol(), by specifying the desired target C data type of the application variable. Refer to "Data Types and Data Conversion" on page 28 for more information.

To unbind a particular column of the result set, use SQLBindCol() with a null pointer for the application variable argument (*TargetValuePtr*). To unbind all of the columns with one function call, use SQLFreeStmt() with an *Option* of SQL_UNBIND.

Applications will perform better if columns are bound, rather than having them retrieved as unbound columns using SQLGetData(). However, an application may be constrained in the amount of long data that it can retrieve

and handle at one time. If this is a concern, then SQLGetData() may be the better choice. See “Using Large Objects” on page 116 for additional techniques to handle long data.

For information on more advanced methods for binding application storage to result set columns, refer to:

- “Retrieving a Result Set into an Array” on page 90
- “Sending/Retrieving Long Data in Pieces” on page 80
- “Column Binding Offsets” on page 93

Processing UPDATE, DELETE and INSERT Statements: If the statement is modifying data (UPDATE, DELETE or INSERT), no action is required, other than the normal check for diagnostic messages. In this case, SQLRowCount() can be used to obtain the number of rows affected by the SQL statement.

If the SQL statement is a Positioned UPDATE or DELETE, it will be necessary to use a *cursor*. A cursor is a moveable pointer to a row in the result table of an active query statement. (This query statement must contain the FOR UPDATE OF clause to ensure that the query is not opened as readonly.) In embedded SQL, cursors names are used to retrieve, update or delete rows. In DB2 CLI, a cursor name is needed only for Positioned UPDATE or DELETE SQL statements as they reference the cursor by name. Furthermore, a cursor name is automatically generated when SQLAllocHandle() is called with a *HandleType* of SQL_HANDLE_STMT.

To update a row that has been fetched, the application uses two statement handles, one for the fetch and one for the update. The application calls SQLGetCursorName() to obtain the cursor name. The application generates the text of a Positioned UPDATE or DELETE, including this cursor name, and executes that SQL statement using a second statement handle. The application cannot reuse the fetch statement handle to execute a Positioned UPDATE or DELETE as it is still in use. You can also define your own cursor name using SQLSetCursorName(), but it is best to use the generated name, since all error messages will reference the generated name, and not the one defined by SQLSetCursorName().

Processing Other Statements: If the statement neither queries nor modifies the data, then there is no further action other than the normal check for diagnostic messages.

Commit or Rollback

A *transaction* is a recoverable unit of work, or a group of SQL statements that can be treated as one atomic operation. This means that all the operations within the group are guaranteed to be completed (committed) or undone

(rolled back), as if they were a single operation. A transaction can also be referred to as a Unit of Work or a Logical Unit of Work. When the transaction spans multiple connections, it is referred to as a Distributed Unit of Work.

DB2 CLI supports two commit modes:

auto-commit
: In auto-commit mode, every SQL statement is a complete transaction, which is automatically committed. For a non-query statement, the commit is issued at the end statement execution. For a query statement, the commit is issued after the cursor has been closed. The application must not start a second query before the cursor of the first query has been closed.

manual-commit
: In manual-commit mode, transactions are started implicitly with the first access to the database using `SQLPrepare()`, `SQLExecDirect()`, `SQLGetTypeInfo()`, or any function that returns a result set, such as those described in "Querying System Catalog Information" on page 65. At this point a transaction has begun, even if the call failed. The transaction ends when you use `SQLEndTran()` to either rollback or commit the transaction. This means that any statements executed (on the same connection) between these are treated as one transaction.

The default commit mode is auto-commit (except when participating in a coordinated transaction, see "Multisite Updates (Two Phase Commit)" on page 50). An application can switch between manual-commit and auto-commit modes by calling `SQLSetConnectAttr()`. Typically, a query-only application may wish to stay in auto-commit mode. Applications that need to perform updates to the database should turn off auto-commit as soon as the database connection has been established.

When multiple connections exist to the same or different databases, each connection has its own transaction. Special care must be taken to call `SQLEndTran()` with the correct connection handle to ensure that only the intended connection and related transaction is affected. It is also possible to rollback or commit all the connections by specifying a valid environment handle, and a NULL connection handle on the `SQLEndTran()` call. Unlike distributed unit of work connections (described in "Multisite Updates (Two Phase Commit)" on page 50), there is no coordination between the transactions on each connection.

When to Call SQLEndTran(): If the application is in auto-commit mode, it never needs to call `SQLEndTran()`. A commit is issued implicitly at the end of each statement execution.

In manual-commit mode, `SQLEndTran()` must be called before calling `SQLDisconnect()`. If Distributed Unit of Work is involved, additional rules may apply, refer to "Multisite Updates (Two Phase Commit)" on page 50 for details.

It is recommended that an application that performs updates should not wait until the disconnect before committing or rolling back the transaction. The other extreme is to operate in auto-commit mode, which is also not recommended as this adds extra processing. Refer to the "Environment, Connection, and Statement Attributes" on page 42 and "SQLSetConnectAttr - Set Connection Attributes" on page 618 for information about switching between auto-commit and manual-commit.

Consider the following when deciding where in the application to end a transaction:

- Each connection has only one outstanding transaction, so keep dependent statements within the same transaction.
- Various resources may be held while you have an outstanding transaction. Ending the transaction will release the resources for use by other users.
- Once a transaction has successfully been committed or rolled back, it is fully recoverable from the system logs. Open transactions are not recoverable.

Effects of calling SQLEndTran():

When a transaction ends:

- All locks on DBMS objects are released, except those that are associated with a held cursor.
- Prepared statements are preserved from one transaction to the next. Once a statement has been prepared on a specific statement handle, it does not need to be re-prepared even after a commit or rollback, provided the statement continues to be associated with the same statement handle.
- Cursor names, bound parameters, and column bindings are maintained from one transaction to the next.
- By default, cursors are preserved after a commit (but not a rollback). In other words, all cursors are by default defined with the WITH HOLD clause (except when connected to SQL/DS, which does not support the WITH HOLD clause, and when the CLI application is running in a Distributed Unit of Work environment, see "Multisite Updates (Two Phase Commit)" on page 50). For information about changing the default behavior, refer to "SQLSetStmtAttr - Set Options Related to a Statement" on page 702.

For more information and an example refer to "SQLEndTran - End Transactions of a Connection" on page 356.

Freeing Statement Handles

Call SQLFreeStmt() to end processing for a particular statement handle. This function can be used to do one or more of the following:

- Unbind all columns of the result set with the exception of the bookmark column if it is used. See "Scrollable Cursors" on page 68 for more information on using bookmarks.

  The SQL_DESC_COUNT field of the application row descriptor (ARD) will also be set to zero in this case. See "Using Descriptors" on page 97 for more information on using descriptors.

- Unbind all parameter markers.

  The SQL_DESC_COUNT field of the application parameter descriptor (APD) will also be set to zero in this case. See "Using Descriptors" on page 97 for more information on using descriptors.

- Close any cursors and discard any pending results (this can also be done using SQLCloseCursor())

Call SQLFreeHandle() with a *HandleType* of SQL_HANDLE_STMT to:

- Drop the statement handle, and release all associated resources

The columns and parameters should always be unbound before using the handle to process a statement with a different number or type of parameters or a different result set; otherwise application programming errors may occur.

Diagnostics

Diagnostics refers to dealing with warning or error conditions generated within an application. There are two levels of diagnostics returned when calling DB2 CLI functions :

- Return Codes
- Detailed Diagnostics (SQLSTATEs, messages, SQLCA)

Each CLI function returns the function return code as a basic diagnostic. Both SQLGetDiagRec() and SQLGetDiagField() functions provide more detailed diagnostic information. The SQLGetSQLCA() function provides access to the SQLCA, if the diagnostic is reported by the data source. This arrangement lets applications handle the basic flow control based on Return Codes, and the SQLSTATES allow determination of the specific causes of failure and specific error handling.

Both SQLGetDiagRec() and SQLGetDiagField() return three pieces of information:

- SQLSTATE

- Native error: if the diagnostic is detected by the data source, this is the SQLCODE; otherwise, this is set to -99999.
- Message text: this is the message text associated with the SQLSTATE.

For the detailed function information and example usage, refer to "SQLGetDiagRec - Get Multiple Fields Settings of Diagnostic Record" on page 477 and "SQLGetDiagField - Get a Field of Diagnostic Data" on page 468.

`SQLGetSQLCA()` returns the SQLCA for access to specific fields, but should never be used as a substitute for `SQLGetDiagRec()` or `SQLGetDiagField()`.

Function Return Codes

The following table lists all possible return codes for DB2 CLI functions. Each function description in "Chapter 5. DB2 CLI Functions" on page 209 lists the possible codes returned for each function.

Table 1. DB2 CLI Function Return Codes

Return Code	Explanation
SQL_SUCCESS	The function completed successfully, no additional SQLSTATE information is available.
SQL_SUCCESS_WITH_INFO	The function completed successfully, with a warning or other information. Call `SQLGetDiagRec()` to receive the SQLSTATE and any other informational messages or warnings. The SQLSTATE will have a class of '01', see "SQLState Cross Reference table" on page 799.
SQL_STILL_EXECUTING	The function is running asynchronously and has not yet completed. The DB2 CLI driver has returned control to the application after calling the function, but the function has not yet finished executing. See "Asynchronous Execution of CLI" on page 138 for complete details.
SQL_NO_DATA_FOUND	The function returned successfully, but no relevant data was found. When this is returned after the execution of an SQL statement, additional information may be available and can be obtained by calling `SQLGetDiagRec()`.
SQL_NEED_DATA	The application tried to execute an SQL statement but DB2 CLI lacks parameter data that the application had indicated would be passed at execute time. For more information, refer to "Sending/Retrieving Long Data in Pieces" on page 80.

Table 1. DB2 CLI Function Return Codes (continued)

Return Code	Explanation
SQL_ERROR	The function failed. Call SQLGetDiagRec() to receive the SQLSTATE and any other error information.
SQL_INVALID_HANDLE	The function failed due to an invalid input handle (environment, connection or statement handle). This is a programming error. No further information is available.

SQLSTATEs

SQLSTATEs are alphanumeric strings of 5 characters (bytes) with a format of ccsss, where cc indicates class and sss indicates subclass. Any SQLSTATE that has a class of:

- '01', is a warning.
- 'HY', is generated by the DB2 CLI or ODBC driver.
- 'IM', is only generated by the ODBC driver manager.

Note: Previous versions of DB2 CLI returned SQLSTATEs with a class of 'S1' rather than 'HY'. To force the CLI driver to return 'S1' SQLSTATEs, the application should set the environment attribute SQL_ATTR_ODBC_VERSION to the value SQL_OV_ODBC2. See "SQLSetEnvAttr - Set Environment Attribute" on page 682 and "Appendix B. Migrating Applications" on page 767 for more information.

DB2 CLI SQLSTATEs include both additional IBM defined SQLSTATEs that are returned by the database server, and DB2 CLI defined SQLSTATEs for conditions that are not defined in the ODBC v3 and ISO SQL/CLI specification. This allows for the maximum amount of diagnostic information to be returned. When running applications in a ODBC environment, it is also possible to receive ODBC defined SQLSTATEs.

Follow these guidelines for using SQLSTATEs within your application:

- Always check the function return code before calling SQLGetDiagRec() to determine if diagnostic information is available.
- Use the SQLSTATEs rather than the native error code.
- To increase your application's portability, only build dependencies on the subset of DB2 CLI SQLSTATEs that are defined by the ODBC v3 and ISO SQL/CLI specification, and return the additional ones as information only. (Dependencies refers to the application making logic flow decisions based on specific SQLSTATEs.)

Note: It may be useful to build dependencies on the class (the first 2 characters) of the SQLSTATEs.

- For maximum diagnostic information, return the text message along with the SQLSTATE (if applicable, the text message will also include the IBM defined SQLSTATE). It is also useful for the application to print out the name of the function that returned the error.

Refer to "SQLState Cross Reference table" on page 799 for a listing and description of the SQLSTATEs explicitly returned by DB2 CLI.

You can use the CLI/ODBC trace facility to gain a better understanding of how your application calls DB2, including any errors that may occur. Refer to "Appendix K. Using the DB2 CLI/ODBC Trace Facility" on page 857 and the CLI/ODBC configuration keyword "TRACE" on page 201.

SQLCA

Embedded applications rely on the SQLCA for all diagnostic information. Although DB2 CLI applications can retrieve much of the same information by using `SQLGetDiagRec()`, there may still be a need for the application to access the SQLCA related to the processing of a statement. (For example, after preparing a statement, the SQLCA will contain the relative cost of executing the statement.) The SQLCA only contains meaningful information if there was an interaction with the data source on the previous request (for example: connect, prepare, execute, fetch, disconnect).

The `SQLGetSQLCA()` function is used to retrieve this structure. Refer to "SQLGetSQLCA - Get SQLCA Data Structure" on page 541 for more information.

`SQLGetSQLCA()` should never be used as a substitute for `SQLGetDiagRec()` or `SQLGetDiagField()`.

Data Types and Data Conversion

When writing a DB2 CLI application it is necessary to work with both SQL data types and C data types. This is unavoidable since the DBMS uses SQL data types, and the application must use C data types. This means the application must match C data types to SQL data types when transferring data between the DBMS and the application (when calling DB2 CLI functions).

To help address this, DB2 CLI provides symbolic names for the various data types, and manages the transfer of data between the DBMS and the application. It will also perform data conversion (from a C character string to

an SQL INTEGER type, for example) if required. To accomplish this, DB2 CLI needs to know both the source and target data type. This requires the application to identify both data types using symbolic names.

C and SQL Data Types

Table 2 list each of the SQL data types, with its corresponding symbolic name, and the default C symbolic name.

SQL Data Type
This column contains the SQL data types as they would appear in an SQL CREATE DDL statement. The SQL data types are dependent on the DBMS.

Symbolic SQL Data Type
This column contains a SQL symbolic names that are defined (in `sqlcli.h`) as an integer value. These values are used by various functions to identify the SQL data types listed in the first column. Refer to "Example" on page 340 for an example using these values.

Default C Symbolic Data Type
This column contains C symbolic names, also defined as an integer values. These values are used in various functions arguments to identify the C data type as shown in Table 3 on page 31. The symbolic names are used by various functions, (such as `SQLBindParameter()`, `SQLGetData()`, `SQLBindCol()`, etc.) to indicate the C data types of the application variables. Instead of explicitly identifying the C data type when calling these functions, SQL_C_DEFAULT can be specified instead, and DB2 CLI will assume a default C data type based on the SQL data type of the parameter or column as shown by this table. For example, the default C data type of SQL_DECIMAL is SQL_C_CHAR.

Table 2. SQL Symbolic and Default Data Types

SQL Data Type	Symbolic SQL Data Type	Default Symbolic C Data Type
BIGINT	SQL_BIGINT	SQL_C_BIGINT
BLOB	SQL_BLOB	SQL_C_BINARY
BLOB LOCATOR [a]	SQL_BLOB_LOCATOR	SQL_C_BLOB_LOCATOR
CHAR	SQL_CHAR	SQL_C_CHAR
CHAR FOR BIT DATA [b]	SQL_BINARY	SQL_C_BINARY
CLOB	SQL_CLOB	SQL_C_CHAR
CLOB LOCATOR [a]	SQL_CLOB_LOCATOR	SQL_C_CLOB_LOCATOR
DATE	SQL_TYPE_DATE [d]	SQL_C__TYPE_DATE [d]
DBCLOB	SQL_DBCLOB	SQL_C_DBCHAR

Table 2. SQL Symbolic and Default Data Types (continued)

SQL Data Type	Symbolic SQL Data Type	Default Symbolic C Data Type
DBCLOB LOCATOR [a]	SQL_DBCLOB_LOCATOR	SQL_C_DBCLOB_LOCATOR
DECIMAL	SQL_DECIMAL	SQL_C_CHAR
DOUBLE	SQL_DOUBLE	SQL_C_DOUBLE
FLOAT	SQL_FLOAT	SQL_C_DOUBLE
GRAPHIC	SQL_GRAPHIC	SQL_C_DBCHAR
INTEGER	SQL_INTEGER	SQL_C_LONG
LONG VARCHAR [b]	SQL_LONGVARCHAR	SQL_C_CHAR
LONG VARCHAR FOR BIT DATA [b]	SQL_LONGVARBINARY	SQL_C_BINARY
LONG VARGRAPHIC [b]	SQL_LONGVARGRAPHIC	SQL_C_DBCHAR
NUMERIC [c]	SQL_NUMERIC [c]	SQL_C_CHAR
REAL	SQL_REAL	SQL_C_FLOAT
SMALLINT	SQL_SMALLINT	SQL_C_SHORT
TIME	SQL_TYPE_TIME [d]	SQL_C_TYPE_TIME [d]
TIMESTAMP	SQL_TYPE_TIMESTAMP [d]	SQL_C_TYPE_TIMESTAMP [d]
VARCHAR	SQL_VARCHAR	SQL_C_CHAR
VARCHAR FOR BIT DATA [b]	SQL_VARBINARY	SQL_C_BINARY
VARGRAPHIC	SQL_VARGRAPHIC	SQL_C_DBCHAR

a LOB locator types are not persistent SQL data types, (columns can not be defined with a locator type, they are only used to describe parameter markers, or to represent a LOB value), refer to “Using Large Objects” on page 116

b LONG data types and FOR BIT DATA data types should be replaced by an appropriate LOB types whenever possible.

c NUMERIC is a synonym for DECIMAL on DB2 for MVS/ESA, DB2 for VSE & VM and DB2 Universal Database.

d See “Appendix B. Migrating Applications” on page 767 for information on what data type was used in previous releases.

Note: The data types, DATE, DECIMAL, NUMERIC, TIME, and TIMESTAMP cannot be transferred to their default C buffer types without a conversion.

Table 3 on page 31 shows the generic type definitions for each symbolic C type.

C Symbolic Data Type

This column contains C symbolic names, defined as integer values. These values are used in various functions arguments to identify the C data type shown in the last column. Refer to "Example" on page 236 for an example using these values.

C Type

This column contains C defined types, defined in `sqlcli.h` using a C `typedef` statement. The values in this column should be used to declare all DB2 CLI related variables and arguments, in order to make the application more portable. Refer to Table 5 on page 33 for a list of additional symbolic data types used for function arguments.

Base C type

This column is shown for reference only, all variables and arguments should be defined using the symbolic types in the previous column. Some of the values are C structures that are described in Table 4 on page 32.

Table 3. C Data Types

C Symbolic Data Type	C Type	Base C type
SQL_C_CHAR	SQLCHAR	unsigned char
SQL_C_BIT	SQLCHAR	unsigned char or char (Value 1 or 0)
SQL_C_TINYINT	SQLSCHAR	signed char (Range -128 to 127)
SQL_C_SHORT	SQLSMALLINT	short int
SQL_C_LONG	SQLINTEGER	long int
SQL_C_DOUBLE	SQLDOUBLE	double
SQL_C_FLOAT	SQLREAL	float
SQL_C_SBIGINT	SQLBIGINT	_int64
SQL_C_UBIGINT	SQLBIGINT	unsigned _int64
SQL_C_NUMERIC [c]	SQL_NUMERIC_STRUCT	see Table 4 on page 32
SQL_C_TYPE_DATE [b]	DATE_STRUCT	see Table 4 on page 32
SQL_C_TYPE_TIME [b]	TIME_STRUCT	see Table 4 on page 32
SQL_C_TYPE_TIMESTAMP [b]	TIMESTAMP_STRUCT	see Table 4 on page 32
SQL_C_CLOB_LOCATOR [a]	SQLINTEGER	long int
SQL_C_BINARY	SQLCHAR	unsigned char
SQL_C_BLOB_LOCATOR [a]	SQLINTEGER	long int
SQL_C_DBCHAR	SQLDBCHAR	wchar_t
SQL_C_DBCLOB_LOCATOR	SQLINTEGER	long int

Table 3. C Data Types (continued)

C Symbolic Data Type	C Type	Base C type

a LOB Locator Types.

b See "Appendix B. Migrating Applications" on page 767 for information on what data type was used in previous releases.

c 32-bit Windows only.

Note: fcSQL file reference data types (used in embedded SQL) are not needed in DB2 CLI, refer to "Using Large Objects" on page 116

Table 4. C Structures

C Type	Generic Structure	Windows Structure
DATE_STRUCT	typedef struct DATE_STRUCT { SQLSMALLINT year; SQLUSMALLINT month; SQLUSMALLINT day; } DATE_STRUCT;	typedef struct tagDATE_STRUCT { SWORD year; UWORD month; UWORD day; } DATE_STRUCT;
TIME_STRUCT	typedef struct TIME_STRUCT { SQLUSMALLINT hour; SQLUSMALLINT minute; SQLUSMALLINT second; } TIME_STRUCT;	typedef struct tagTIME_STRUCT { UWORD hour; UWORD minutes; UWORD second; } TIME_STRUCT;
TIMESTAMP_STRUCT	typedef struct TIMESTAMP_STRUCT { SQLUSMALLINT year; SQLUSMALLINT month; SQLUSMALLINT day; SQLUSMALLINT hour; SQLUSMALLINT minute; SQLUSMALLINT second; SQLINTEGER fraction; } TIMESTAMP_STRUCT;	typedef struct tagTIMESTAMP_STRUCT { SWORD year; UWORD month; UWORD day; UWORD hour; UWORD minute; UWORD second; UDWORD fraction; } TIMESTAMP_STRUCT;
SQL_NUMERIC_STRUCT	(None. Windows 32-bit only)	typedef struct tagSQL_NUMERIC_STRUCT { SQLCHAR precision; SQLCHAR scale; SQLCHAR sign; [a] SQLCHAR val[SQL_MAX_NUMERIC_LEN];[b] [c] } SQL_NUMERIC_STRUCT;

Table 4. C Structures (continued)

C Type	Generic Structure	Windows Structure

Refer to Table 5 for more information on the SQLUSMALLINT C data type.

a Sign field: 1 = positive, 2 = negative

b A number is stored in the val field of the SQL_NUMERIC_STRUCT structure as a scaled integer, in little endian mode (the leftmost byte being the least-significant byte). For example, the number 10.001 base 10, with a scale of 4, is scaled to an integer of 100010. Because this is 186AA in hexadecimal format, the value in SQL_NUMERIC_STRUCT would be "AA 86 01 00 00 ... 00", with the number of bytes defined by the SQL_MAX_NUMERIC_LEN #define.

c The precision and scale fields of the SQL_C_NUMERIC data type are never used for input from an application, only for output from the driver to the application. When the driver writes a numeric value into the SQL_NUMERIC_STRUCT, it will use its own default as the value for the precision field, and it will use the value in the SQL_DESC_SCALE field of the application descriptor (which defaults to 0) for the scale field. An application can provide its own values for precision and scale by setting the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the application descriptor.

Other C Data Types

As well as the data types that map to SQL data types, there are also C symbolic types used for other function arguments, such as pointers and handles. Both the generic and ODBC data types are shown below.

Table 5. C Data Types and Base C Data Types

Defined C Type	Base C Type	Typical Usage
SQLPOINTER	void *	Pointers to storage for data and parameters.
SQLHANDLE	long int	Handle used to reference all 4 types of handle information.
SQLHENV	long int	Handle referencing environment information.
SQLHDBC	long int	Handle referencing database connection information.
SQLHSTMT	long int	Handle referencing statement information.
SQLUSMALLINT	unsigned short int	Function input argument for unsigned short integer values.
SQLUINTEGER	unsigned long int	Function input argument for unsigned long integer values.
SQLRETURN	short int	Return code from DB2 CLI functions.

Versions of DB2 CLI prior to Version 2.1:

- Defined SQLRETURN as a long (32-bit) integer.
- Used SQLSMALLINT and SQLINTEGER instead of SQLUSMALLINT and SQLUINTEGER (signed instead of unsigned). Refer to "Appendix B. Migrating Applications" on page 767 for more information.

Data Conversion

As mentioned previously, DB2 CLI manages the transfer and any required conversion of data between the application and the DBMS. Before the data transfer actually takes place, the source, target or both data types are indicated when calling SQLBindParameter(), SQLBindCol() or SQLGetData(). These functions use the symbolic type names shown in Table 2 on page 29, to identify the data types involved.

For example, to bind a parameter marker that corresponds to an SQL data type of DECIMAL(5,3), to an application's C buffer type of double, the appropriate SQLBindParameter() call would look like:

```
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_DOUBLE,
                  SQL_DECIMAL, 5, 3, double_ptr, 0, NULL);
```

Table 2 on page 29 shows only the default data conversions. The functions mentioned in the previous paragraph can be used to convert data to other types, but not all data conversions are supported or make sense. Table 6 on page 35 shows all the conversions supported by DB2 CLI.

The first column in Table 6 on page 35 contains the data type of the SQL data type. The remaining columns represent the C data types. If the C data type columns contains:

D The conversion is supported and is the default conversion for the SQL data type.

X all IBM DBMSs support the conversion,

blank no IBM DBMS supports the conversion.

As an example, the table indicates that a CHAR (or a C character string as indicated in Table 6 on page 35) can be converted into a SQL_C_LONG (a signed long). In contrast, a LONGVARCHAR cannot be converted to a SQL_C_LONG.

Refer to "Appendix F. Data Conversion" on page 817 for information about the required formats and the results of converting between data types.

There are rules specified in the *SQL Reference* for limits on precision and scale, as well as truncation and rounding rules for type conversions. These rules apply in DB2 CLI, with the following exception: truncation of values to the right of the decimal point for numeric values may return a truncation warning, whereas truncation to the left of the decimal point returns an error. In cases of error, the application should call SQLGetDiagRec() to obtain the SQLSTATE and additional information on the failure. When moving and

converting floating point data values between the application and DB2 CLI, no correspondence is guaranteed to be exact as the values may change in precision and scale.

Table 6. Supported Data Conversions

SQL Data Type	SQL_C_CHAR	SQL_C_LONG	SQL_C_SHORT	SQL_C_TINYINT	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_TYPE_DATE	SQL_C_TYPE_TIME	SQL_C_TYPE_TIMESTAMP	SQL_C_BINARY	SQL_C_BIT	SQL_C_DBCHAR	SQL_C_CLOB_LOCATOR	SQL_C_BLOB_LOCATOR	SQL_C_DBCLOB_LOCATOR	SQL_C_BIGINT	SQL_C_NUMERIC
BLOB	X									D				X			
CHAR	D	X	X	X	X	X	X	X	X	X	X					X	X
CLOB	D									X			X				
DATE	X						D		X								
DBCLOB										X		D			X		
DECIMAL	D	X	X	X	X	X				X	X					X	X
DOUBLE	X	X	X	X	X	D					X					X	X
FLOAT	X	X	X	X	X	D					X					X	X
GRAPHIC	X											D					
INTEGER	X	D	X	X	X	X					X					X	X
LONG VARCHAR	D									X							
LONG VARGRAPHIC	X									X		D					
NUMERIC	D	X	X	X	X	X					X						X
REAL	X	X	X	X	D	X					X						X
SMALLINT	X	X	D	X	X	X					X					X	X

Table 6. Supported Data Conversions (continued)

SQL Data Type	SQL_C_CHAR	SQL_C_LONG	SQL_C_SHORT	SQL_C_TINYINT	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_TYPE_DATE	SQL_C_TYPE_TIME	SQL_C_TYPE_TIMESTAMP	SQL_C_BINARY	SQL_C_BIT	SQL_C_DBCHAR	SQL_C_CLOB_LOCATOR	SQL_C_BLOB_LOCATOR	SQL_C_DBCLOB_LOCATOR	SQL_C_BIGINT	SQL_C_NUMERIC
BIGINT	X	X	X	X	X	X				X	X					D	X
TIME	X							D	X								
TIMESTAMP	X						X	X	D								
VARCHAR	D	X	X	X	X	X	X	X	X	X	X					X	X
VARGRAPHIC	X											D					

Note:

- Data is not converted to LOB Locator types, rather locators represent a data value, refer to “Using Large Objects” on page 116 for more information.
- REAL is not supported by DB2 Universal Database.
- NUMERIC is a synonym for DECIMAL on DB2 for MVS/ESA, DB2 for VSE & VM, and DB2 Universal Database.
- SQL_C_NUMERIC is only available on 32-bit Windows operating systems.

Working with String Arguments

The following conventions deal with the various aspects of working with string arguments in DB2 CLI functions.

Length of String Arguments

Input string arguments have an associated length argument. This argument indicates either the exact length of the argument (not including the null terminator), the special value SQL_NTS to indicate a null-terminated string, or

SQL_NULL_DATA to pass a NULL value. If the length is set to SQL_NTS, DB2 CLI will determine the length of the string by locating the null terminator.

Output string arguments have two associated length arguments: an input length argument to specify the length of the allocated output buffer, and an output length argument to return the actual length of the string returned by DB2 CLI. The returned length value is the total length of the string available for return, regardless of whether it fits in the buffer or not.

For SQL column data, if the output is a null value, SQL_NULL_DATA is returned in the length argument and the output buffer is untouched. The descriptor field SQL_DESC_INDICATOR_PTR is set to SQL_NULL_DATA if the column value is a null value. For more information, including which other fields are set, see "SQL_DESC_INDICATOR_PTR" on page 667 in `SQLSetDescField()`.

If a function is called with a null pointer for an output length argument, DB2 CLI will not return a length, and assumes that the data buffer is large enough to hold the data. When the output data is a NULL value, DB2 CLI can not indicate that the value is NULL. If it is possible that a column in a result set can contain a NULL value, a valid pointer to the output length argument must always be provided. It is highly recommended that a valid output length argument always be used.

Null-Termination of Strings

By default, every character string that DB2 CLI returns is terminated with a null terminator (hex 00), except for strings returned from graphic and DBCLOB data types into SQL_C_CHAR application variables. Graphic and DBCLOB data types that are retrieved into SQL_C_DBCHAR application variables are null terminated with a double byte null terminator. This requires that all buffers allocate enough space for the maximum number of bytes expected, plus the null-terminator.

It is also possible to use `SQLSetEnvAttr()` and set an environment attribute to disable null termination of variable length output (character string) data. In this case, the application allocates a buffer exactly as long as the longest string it expects. The application must provide a valid pointer to storage for the output length argument so that DB2 CLI can indicate the actual length of data returned; otherwise, the application will not have any means to determine this. The DB2 CLI default is to always write the null terminator.

It is possible, using the PATCH1 CLI/ODBC configuration keyword, to force DB2 CLI to null terminate graphic and DBCLOB strings. This keyword can be set from the CLI/ODBC Settings notebook accessible from the Client

Configuration Assistant (CCA). Refer to the “Platform Specific Details for CLI/ODBC Access” on page 151. The *Configure the CLI/ODBC Driver* section for your platform will provide the steps required to set the keywords. The description of PATCH1 in “Configuration Keywords” on page 164 includes how to find the setting required to force the null termination of graphic and DBCLOB strings.

String Truncation

If an output string does not fit into a buffer, DB2 CLI will truncate the string to the size of the buffer, and write the null terminator. If truncation occurs, the function will return SQL_SUCCESS_WITH_INFO and an SQLSTATE of **01**004 indicating truncation. The application can then compare the buffer length to the output length to determine which string was truncated.

For example, if `SQLFetch()` returns SQL_SUCCESS_WITH_INFO, and an SQLSTATE of **01**004, at least one of the buffers bound to a column is too small to hold the data. For each buffer that is bound to a column, the application can compare the buffer length with the output length and determine which column was truncated.

Interpretation of Strings

Normally, DB2 CLI interprets string arguments in a case-sensitive manner and does not trim any spaces from the values. The one exception is the cursor name input argument on the `SQLSetCursorName()` function, where if the cursor name is not delimited (enclosed by double quotes) the leading and trailing blanks are removed and case is ignored.

Querying Environment and Data Source Information

There are many situations where an application requires information about the characteristics and capabilities of the current DB2 CLI driver or the data source that it is connected to. DB2 CLI provides a number of functions that return this information.

There are two common situations where the application requires this information:

- Application displays information for the user. Information such as the data source name and version, or the version of the DB2 CLI driver might be displayed at connect time, or as part of the error reporting process.
- Generic applications that are written to adapt and take advantage of facilities that may be available from some, but not all database servers.

The following DB2 CLI functions provide data source specific information:

- “SQLDataSources - Get List of Data Sources” on page 332

- “SQLGetFunctions - Get Functions” on page 483
- “SQLGetInfo - Get General Information” on page 489
- “SQLGetTypeInfo - Get Data Type Information” on page 555

Querying Environment Information Example

The `getinfo.c` sample, shown in “Example” on page 531 generates the following output when connected to DB2.

```
       Server Name: SAMPLE
     Database Name: SAMPLE
     Instance Name: db2inst1
         DBMS Name: DB2/6000
      DBMS Version: 05.00.0000
    CLI Driver Name: libdb2.a
 CLI Driver Version: 05.00.0000
 ODBC SQL Conformance Level: Extended Grammar
```

Chapter 3. Using Advanced Features

This section covers a series of advanced tasks.

This section does not cover features that are provided generally by dynamic SQL, such as User Defined Functions or Triggers. Refer to the *SQL Reference* for a complete description of SQL language supported.

Environment, Connection, and Statement Attributes

Environments, connections, and statements each have a defined set of attributes (or options). All attributes can be queried by the application, but only some attributes can be changed from their default values. By changing attribute values, the application can change the behavior of DB2 CLI.

An environment handle has attributes which affect the behavior of DB2 CLI functions under that environment. The application can specify the value of an attribute by calling `SQLSetEnvAttr()` and can obtain the current attribute value by calling `SQLGetEnvAttr()`. `SQLSetEnvAttr()` can only be called before connection handles have been allocated.

A connection handle has attributes which affect the behavior of DB2 CLI functions under that connection. Of the attributes that can be changed:

- Some can be set any time once the connection handle is allocated.
- Some can be set only before the actual connection has been established.
- Some can be set only after the connection has been established.

- Some can be set after the connection has been established, but only while there are no outstanding transactions or open cursors.

The application can change the value of connection attributes by calling SQLSetConnectAttr() and can obtain the current value of an attribute by calling SQLGetConnectAttr(). An example of a connection attribute which can be set any time after a handle is allocated is the auto-commit option introduced in "Commit or Rollback" on page 22. For complete details on when each attribute can be set, refer to "SQLSetConnectAttr - Set Connection Attributes" on page 618.

A statement handle has attributes which affect the behavior of CLI functions executed using that statement handle. Of the statement attributes that can be changed:

- Some attributes can be set, but currently can be set to only one specific value.
- Some attributes can be set any time after the statement handle has been allocated.
- Some attributes can only be set if there is no open cursor on that statement handle.

The application can specify the value of any settable statement attribute by calling SQLSetStmtAttr(), and can obtain the current value of an attribute by calling SQLGetStmtAttr(). For complete details on when each attribute can be set, refer to "SQLSetStmtAttr - Set Options Related to a Statement" on page 702.

The SQLSetConnectAttr() function cannot be used to set statement attributes. This was supported in versions of DB2 CLI prior to version 5; see "Setting a Subset of Statement Attributes using SQLSetConnectAttr()" on page 770 for details.

Many applications use just the default attribute settings; however, there may be situations where some of these defaults are not suitable for a particular user of the application. DB2 CLI provides end users with two methods to change some of these default values at run time. The first method is to specify the new default attribute value(s) in the connection string input to the SQLDriverConnect() and SQLBrowseConnect() functions. The second method involves the specification of the new default attribute value(s) in a DB2 CLI initialization file.

The DB2 CLI initialization file can be used to change default values for all DB2 CLI applications on that workstation. This may be the end user's only means of changing the defaults if the application does not provide a means for the user to provide default attribute values in the SQLDriverConnect()

connection string. Default attribute values that are specified on `SQLDriverConnect()` override the values in the DB2 CLI initialization file for that particular connection. For information on how the end user can use the DB2 CLI initialization file as well as for a list of changeable defaults, refer to "DB2 CLI/ODBC Configuration Keyword Listing" on page 164.

The mechanisms for changing defaults are intended for end user tuning; application developers must use the appropriate set-attribute function. If an application does call a set-attribute or option function with a value different from the initialization file or the connection string specification, then the initial default value is overridden and the new value takes effect.

The attributes that can be changed are listed in the detailed function descriptions of the *set* attribute or option functions, see "Chapter 5. DB2 CLI Functions" on page 209. The readonly options (if any exist) are listed with the detailed function descriptions of the *get* attribute or option functions.

For information on some commonly used attributes, refer to "Appendix A. Programming Hints and Tips" on page 757.

The diagram below shows the addition of the attribute functions to the basic connect scenario.

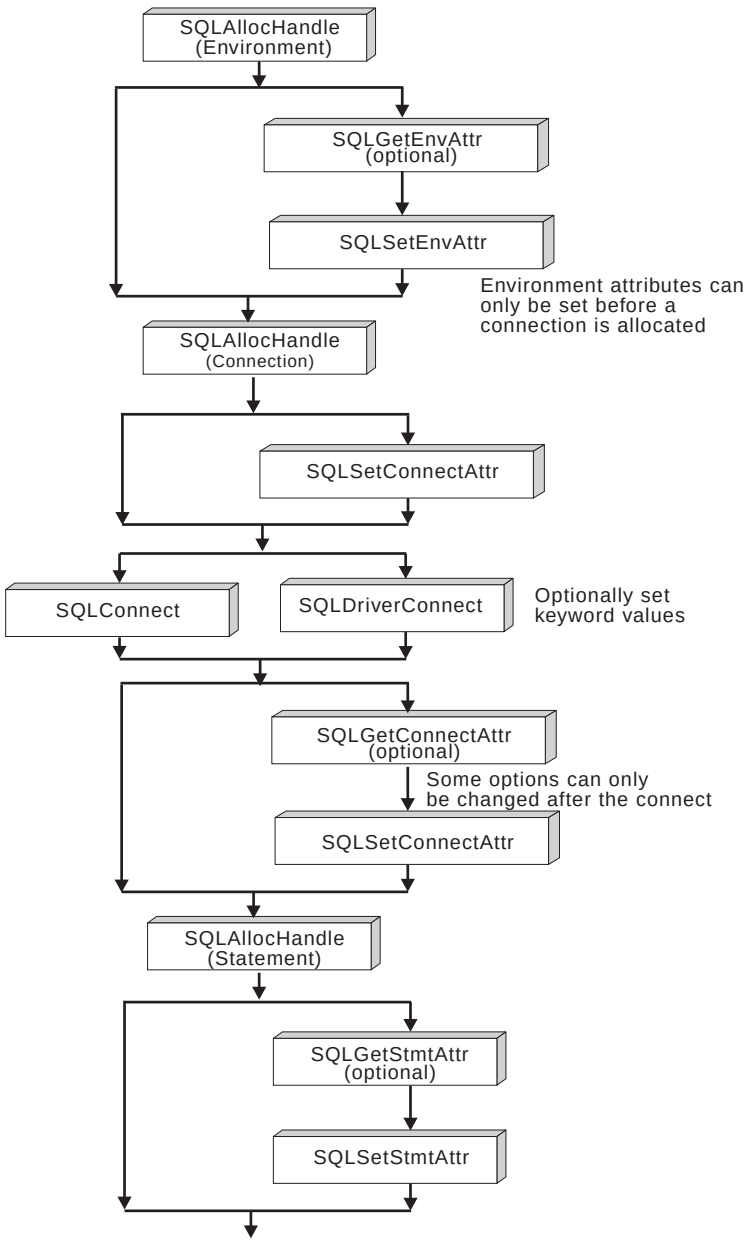


Figure 4. Setting and Retrieving Attributes (Options)

Writing Multi-Threaded Applications

DB2 CLI supports concurrent execution of threads on the following platforms:

- AIX
- HP UX-11
- OS/2
- Silicon Graphics IRIX
- Sun Solaris
- 32-bit Windows

On any other platform that supports threads, DB2 CLI is guaranteed to be thread safe by serializing all calls to DB2 CLI. In other words, DB2 CLI is always reentrant.

Note: If you are writing applications that use DB2 CLI calls and either embedded SQL or DB2 API calls, see "Multi-Threaded Mixed Applications" on page 49.

Concurrent execution means that two threads can run independently of each other (on a multi-processor machine they may run simultaneously). For example, an application could implement a database-to-database copy in the following way:

- One thread connects to database A and uses `SQLExecute()` and `SQLFetch()` calls to read data from one connection into a shared application buffer.
- The other thread connects to database B and concurrently reads from the shared buffer and insert the data into database B.

In contrast, if DB2 CLI serializes all function calls, only one thread may be executing a DB2 CLI function at a time. All other threads would have to wait until the current thread is done before it would get a chance to execute.

When to Use Multiple Threads

The most common reason to create another thread in a DB2 CLI application is so that a thread other than the one executing can be used to call `SQLCancel()` (to cancel a long running query for example).

Note: This method should be used for any platform which supports threads rather than using the asyncronous SQL model (which was designed for non-threaded operating systems such a Windows 3.1). If your application cannot make use of multi-threading then see "Asynchronous Execution of CLI" on page 138.

Most GUI based applications use threads in order to ensure that user interaction can be handled on a higher priority thread than other application tasks. The application can simply delegate one thread to run all DB2 CLI functions (with the exception of SQLCancel()). In this case there are no thread-related application design issues since only one thread will be accessing the data buffers that are used to interact with DB2 CLI.

Applications that use multiple connections, and are executing statements that may take some time to execute, should consider executing DB2 CLI functions on multiple threads to improve throughput. Such an application should follow standard practices for writing any multi-thread application, most notable those concerning sharing data buffers. The following section discusses in more detail what DB2 CLI guarantees and what the application must guarantee in order to write a more complex multi-threaded application.

Programming Tips

Any resource allocated by DB2 CLI is guaranteed to be thread-safe. This is accomplished by using either a shared global or connection specific semaphore. At any one time, only one thread can be executing a DB2 CLI function that accepts an environment handle as input. All other functions that accept a connection handle, (or a statement or descriptor allocated on that connection handle), will be serialized on the connection handle.

This means that once a thread starts executing a function with a connection handle, or child of a connection handle, any other thread will block and wait for the executing thread to return. The one exception to this is SQLCancel(), which must be able to cancel a statement currently executing on another thread. For this reason, the most natural design is to map one thread per connection, plus one thread to handle SQLCancel() requests. Each thread can then execute independently of the others.

As an example, if a thread is using a handle in one thread, and another thread frees that handle between function calls, the next attempt to use that handle would result in a return code of SQL_INVALID_HANDLE.

Note: This only applies for DB2 CLI applications. ODBC applications may trap since the handle in this case is a pointer and the pointer may no longer be valid if another thread has freed it. For this reason, it is best to follow the model below.

Note: There may be platform or compiler specific link options required for multi-threaded applications. Refer to the *Application Building Guide* for complete details.

Sample Application Model

The following is intended as an example:

- Designate a master thread which allocates:
  - *m* ″child″ threads
  - *n* connection handles
- Each task that requires a connection is executed by one of the child threads, and is given one of the *n* connections by the master thread.
- Each connection is marked as in use by the master thread until the child thread returns it to the connection pool.
- An `SQLCancel()` request is handled by the master thread.

This model allows the master thread to have more threads than connections if the threads are also used to perform non-SQL related tasks, or more connections than threads if the application wants to maintain a pool of active connections to various databases, but limit the number of active tasks.

Most importantly, this ensures that two threads are not trying to use the same connection or statement handle at any one time. Although DB2 CLI controls access to its resources, the application resources such as bound columns and parameter buffers are not controlled by DB2 CLI, and the application must guarantee that a pointer to a buffer is not being used by two threads at any one time. Any deferred arguments must remain valid until the column or parameter has been unbound.

If it is necessary for two threads to share a data buffer, the application must implement some form of synchronization mechanism. For example, in the database-to-database copy scenario mentioned above, the use of the shared buffer must be synchronized by the application.

Application Deadlocks

The application must be aware of the possibility of creating deadlock situations with shared resources in the database and the application.

DB2 can detect deadlocks at the server and rollback one or more transactions to resolve them. An application may still deadlock if:

- two thread are connected to the same database, and
- one thread is holding an application resource and is waiting for a database resource, and
- the other thread has a lock on the database resource while waiting for the application resource.

In this case the DB2 Server is only going to see a lock, not a deadlock, and unless the database LOCKTIMEOUT configuration setting is changed, the application will wait forever.

The model suggested above avoids this problem by not sharing application resources between threads once a thread starts executing on a connection.

Problems With Existing Multi-Threaded Applications

It is possible that an existing mulit-threaded DB2 CLI application ran successfully using the serialized version of DB2 CLI (prior to version 5), yet suffers synchronization problems when run using DB2 CLI version 5 or later.

In this case the DISABLEMULTITHREAD CLI/ODBC configuration keyword can be set to 1 in order to force DB2 CLI to serialize all function calls. If this is required, the application should be analyzed and corrected.

Multi-Threaded Mixed Applications

It is possible for a multi-threaded application to mix CLI calls with DB2 API calls and embedded SQL. Which type of call comes first determines the best way to organize the application:

DB2 CLI Calls First

The DB2 CLI driver automatically calls the DB2 context APIs to allocate and manage contexts for the application. This means that any application that calls `SQLAllocEnv()` before calling any other DB2 API or embedded SQL will be initialized with the context type set to SQL_CTX_MULTI_MANUAL.

In this case the application should allow DB2 CLI to allocate and manage all contexts. Use DB2 CLI to allocate all connection handles and to perform all connections. Call the `SQLSetConnect()` function in each thread prior to calling any embedded SQL. DB2 APIs can be called after any DB2 CLI function has been called in the same thread.

DB2 API Calls or Embedded SQL First

The DB2 CLI driver does not automatically call the DB2 context APIs if the application calls the DB2 API or embedded SQL before a CLI function.

This means that any thread that calls a DB2 API or embedded SQL must be attached to a context, otherwise the call will fail with an SQLCODE of SQL1445N. This can be done by calling the DB2 API `sqleAttachToCtx()` which will explicitly attach the thread to a context, or by calling any DB2 CLI function (`SQLSetConnection()` for example).

In this case, the application must explicitly manage all contexts.

Use the context APIs to allocate and attach to contexts prior to calling DB2 CLI functions (`SQLAllocEnv()` will use the existing context as the default context). Use the SQL_ATTR_CONN_CONTEXT connection attribute to explicitly set the context that each DB2 CLI connection should use.

See "Appendix B. Migrating Applications" on page 767 for details on running existing mixed applications.

Multisite Updates (Two Phase Commit)

The transaction scenario described in "Connecting to One or More Data Sources" on page 12 portrays an application which interacts with only one database server in a transaction. Even though concurrent connections allow for concurrent transactions, the different transactions are not coordinated.

With multisite update, also known as Distributed Unit of Work (DUOW), two phase commit (2PC), and Coordinated Distributed Transactions, an application is able to update data in multiple remote database servers with guaranteed integrity.

A typical banking transaction is a good example of a multisite update. Consider the transfer of money from one account to another in a different database server. In such a transaction it is critical that the updates that implement the debit operation on one account do not get committed unless the updates required to process the 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

Some multisite updates involve the use of a Transaction Manager to coordinate two-phase commit among multiple databases. For detailed description of multisite updates, refer to the *Administration Guide*. This section describes how DB2 CLI applications can be written to use various transaction managers:

- "DB2 as Transaction Monitor" on page 51
- "Microsoft Transaction Server (MTS) as Transaction Monitor" on page 57
- "Process-based XA-Compliant Transaction Program Monitor (XA TP)" on page 64
- "Host and AS/400 Database Servers" on page 65

DB2 as Transaction Monitor

DB2 CLI/ODBC applications can use DB2 itself as the Transaction Manager (DB2 TM) to coordinate distributed transactions against all IBM database servers. Please see the *Administration Guide* for more details about the requirements and capabilities of using DB2 as the transaction manager.

Configuration - DB2 as Transaction Monitor

The DB2 Transaction Manager must be set up according to the information in the *Administration Guide*.

To use DB2 as the transaction manager in CLI/ODBC applications, the following CLI/ODBC configuration keywords must be set as follows:

```
[COMMON]
DISABLEMULTITHREAD = 1
CONNECTTYPE=2
SYNCPOINT=2
```

For more information about setting the CLI/ODBC configuration keywords, see "Platform Specific Details for CLI/ODBC Access" on page 151.

Two of the above configuration keywords can also be set using the following environment attributes:

- SQL_ATTR_CONNECT_TYPE to SQL_COORDINATED_TRANS
- SQL_ATTR_SYNCPOINT to SQL_TWOPHASE

See `SQLSetEnvAttr()` for more details. There is no appropriate environment attribute for the DISABLEMULTITHREAD keyword which must therefore still be set to 1 in the [COMMON] section of the db2cli.ini file.

Because the DISABLEMULTITHREAD keyword must appear in the [COMMON] section of the db2cli.ini file it impacts all connections to all datasources from that client instance. This means that a DB2 client instance can only support process-based CLI applications or thread-based CLI applications, but not both

Programming Considerations

You must set the DB2 CLI/ODBC configuration keyword DISABLEMULTITHREAD to 1. This indicates that the DB2 CLI/ODBC driver will use a single DB2 context for all connections made by the application process. All database requests will serialized at the process level.

The environment attribute SQL_ATTR_CONNECTTYPE controls whether the application is to operate in a coordinated or uncoordinated distributed environment. The two possible values for this attribute are:

- SQL_CONCURRENT_TRANS - supports the single database per transaction semantics described in Chapter 2. Multiple concurrent connections to the same database and to different databases are permitted. This is the default.
- SQL_COORDINATED_TRANS - supports the multiple databases per transaction semantics, as discussed below.

All connections within an application must have the same SQL_ATTR_CONNECTTYPE setting. It is recommended that the application set this environment attribute, if necessary, as soon as the environment handle has been created with a call to `SQLAllocHandle()` (with a *HandleType* of SQL_HANDLE_ENV). Since ODBC applications cannot access `SQLSetEnvAttr()`, they must set this using `SQLSetConnectAttr()` before any connection has been established.

Attributes that Govern Multisite Update Semantics: A coordinated transaction means that commits or rollbacks among multiple database connections are coordinated. The SQL_COORDINATED_TRANS setting of the SQL_ATTR_CONNECTTYPE attribute corresponds to the Type 2 CONNECT in IBM embedded SQL and must be considered in conjunction with the SQL_ATTR_SYNC_POINT attribute, which has the following two possible settings:

- SQL_ONEPHASE: One-phase commit is used to commit the work done by each database in a multiple database transaction. To ensure data integrity, each transaction must not have more than one database updated. The first database that has updates performed in a transaction becomes the only updater in that transaction, all other databases accessed are treated as read-only. Any update attempts to these read-only database within this transaction are rejected.
- SQL_TWOPHASE: Two-phase commit is used to commit the work done by each database in a multiple database transaction. This requires the use of a Transaction Manager to coordinate two phase commits amongst the databases that support this protocol. Multiple readers and multiple updaters are allowed within a transaction.

Similar to SQL_ATTR_CONNECTTYPE, it is recommended that the application set this environment attribute, if necessary, as soon as the environment handle has been created with a call to `SQLAllocHandle()` (with a *HandleType* of SQL_HANDLE_ENV). ODBC applications must use `SQLSetConnectAttr()` to set this for each connection handle under the environment before any connections have been established.

All the connections within an application must have the same SQL_ATTR_CONNECTTYPE and SQL_ATTR_SYNC_POINT settings. After the first connection has been established, all subsequent connect types must be the same as the first. Coordinated connections default to manual-commit mode (for discussion on auto-commit mode, see “Commit or Rollback” on page 22).

The function `SQLEndTran()` must be used in a multisite update environment when DB2 is acting as the transaction manager.

Figure 5 on page 54 shows the logical flow of an application executing statements on two SQL_CONCURRENT_TRANS connections (’A’ and ’B’), and indicates the scope of the transactions.

Figure 6 on page 55 shows the same statements being executed on two SQL_COORDINATED_TRANS connections (’A’ and ’B’), and the scope of a coordinated distributed transaction.

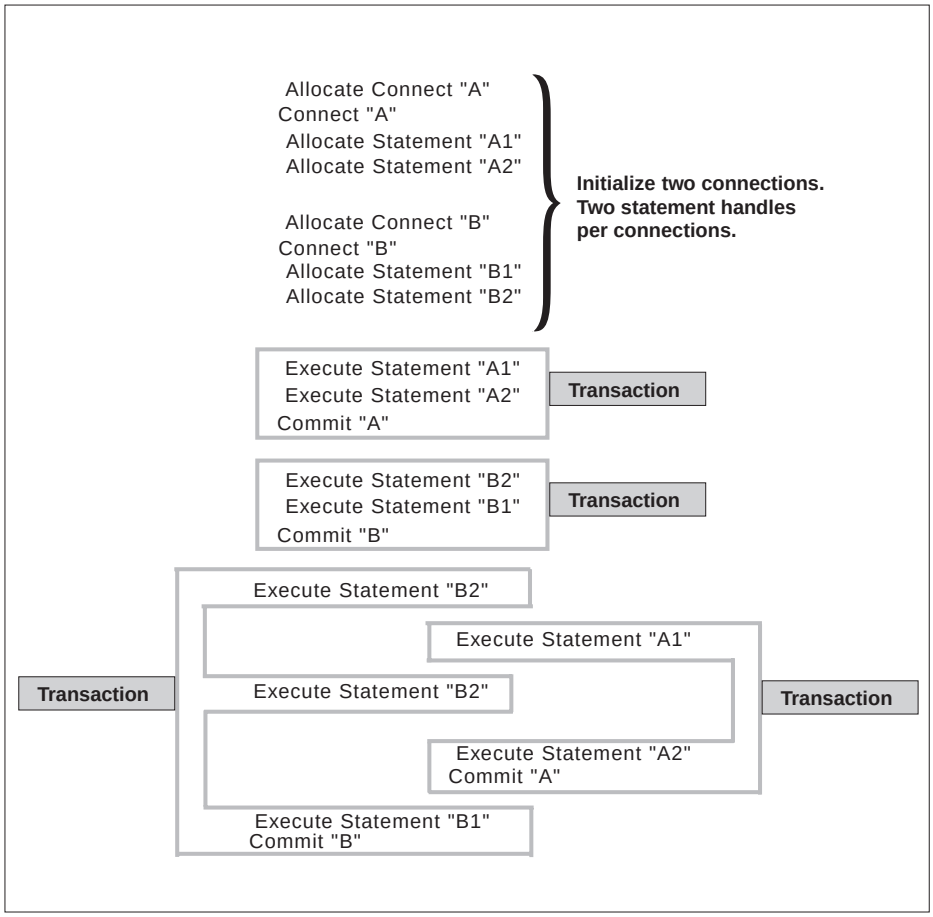


Figure 5. Multiple Connections with Concurrent Transactions

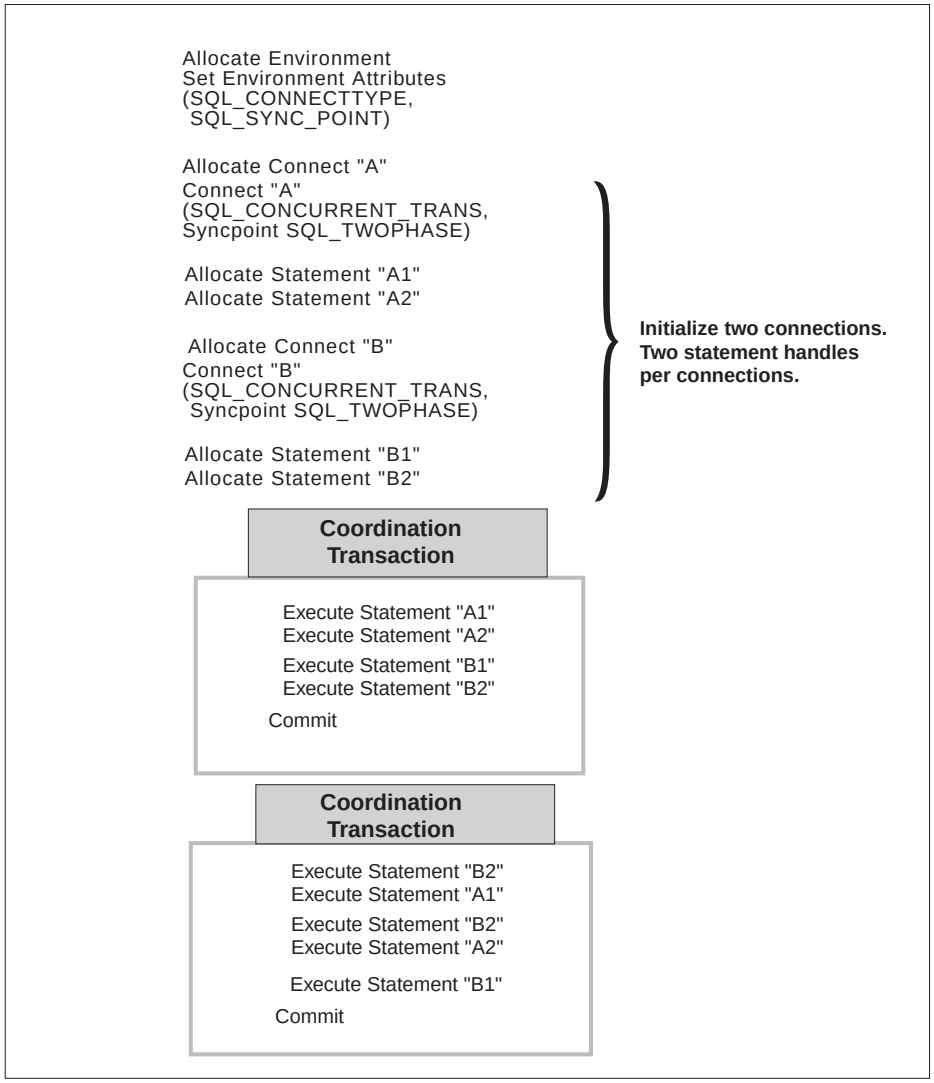


Figure 6. Multiple Connections with Coordinated Transactions

Establishing a Coordinated Transaction Connection: An application can establish coordinated transaction connections by calling the `SQLSetEnvAttr()` function, or by setting the CONNECTTYPE and SYNCPOINT keywords in the DB2 CLI initialization file or in the connection string for `SQLDriverConnect()`. The initialization file is intended for existing applications that do not use the `SQLSetConnectAttr()` function. For information about the keywords, refer to "DB2 CLI/ODBC Configuration Keyword Listing" on page 164.

An application cannot have a mixture of concurrent and coordinated connections, the type of the first connection will determine the type of all subsequent connections. SQLSetEnvAttr() will return an error if an application attempts to change the connect type while there is an active connection.

Restrictions

Mixing Embedded SQL and CLI/ODBC calls in a multisite update environment is supported, but all the same restrictions of writing mixed applications are imposed. Please see “Mixing Embedded SQL and DB2 CLI” on page 136 for more details.

Sample

The following example connects to two data sources using a SQL_ATTR_CONNECTTYPE set to SQL_COORDINATED_TRANS and SQL_ATTR_SYNC_POINT set to SQL_ONEPHASE.

```
/* From CLI sample duowcon.c */
/* ... */
/* main */
int main( int argc, char * argv[] ) {

    SQLHANDLE henv, hdbc[MAX_CONNECTIONS] ;
    SQLRETURN rc ;

/* ... */

    /* allocate an environment handle */
    SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;

    /*
     Before allocating any connection handles, set Environment wide
     Connect Options
     Set to Connect Type 2, Syncpoint 1
    */
    if ( SQLSetEnvAttr( henv,
                        SQL_CONNECTTYPE,
                        ( SQLPOINTER ) SQL_COORDINATED_TRANS,
                        0
                      ) != SQL_SUCCESS ) {
       printf( ">---ERROR while setting Connect Type 2 -------------\n" ) ;
       return( SQL_ERROR ) ;
    }
/* ... */
    if ( SQLSetEnvAttr( henv,
                        SQL_SYNC_POINT,
                        ( SQLPOINTER ) SQL_ONEPHASE,
                        0
                      ) != SQL_SUCCESS ) {
       printf( ">---ERROR while setting Syncpoint One Phase -------------\n" ) ;
       return( SQL_ERROR ) ;
```

```
    }
/* ... */

    /* Connect to first data source */
    prompted_connect( henv, &hdbc[0] ) ;

    /* Connect to second data source */
    DBconnect( henv, &hdbc[1] ) ;

    /*********   Start Processing Step  ************************/
    /* allocate statement handle, execute statement, etc.       */
    /*********   End Processing Step  *************************/

    /* Disconnect, free handles and exit */
```

Microsoft Transaction Server (MTS) as Transaction Monitor

Applications running under Microsoft Transaction Server (MTS) on Windows NT, Windows 95, and Windows 98 operating systems can use MTS to coordinate two-phase commit with multiple DB2 UDB, host, and AS/400 database servers and other MTS-compliant resource managers.

A new connection attribute has been created in `SQLSetConnectAttr()` to support the Microsoft Transaction Server (MTS). See the information on SQL_ATTR_ENLIST_IN_DTC in "SQLSetConnectAttr - Set Connection Attributes" on page 618.

MTS Software Prerequisites

MTS support requires a Version 5.2, or higher DB2 client, and MTS must be at Version 2.0 with Hotfix 0772 or later.

Configuring Microsoft Transaction Server

DB2 UDB V5.2 and following can be fully integrated with Microsoft Transaction Server (MTS) Version 2.0. Applications running under MTS on Windows 32-bit operating systems can use MTS to coordinate two-phase commit with multiple DB2 UDB, host, and AS/400 database servers, as well as with other MTS-compliant resource managers.

Enabling MTS Support in DB2: Microsoft Transaction Server support is automatically enabled. While you can set the *tp_mon_name* database manager configuration parameter to "MTS", it is not necessary and will be ignored.

Note: Additional technical information may be provided on the IBM web site to assist you with installation and configuration of DB2 MTS support. Set your URL to ″http://www.software.ibm.com/data/db2/library″, and search for a DB2 Universal Database ″Technote″ with the keyword ″MTS″.

MTS Software Prerequisites: MTS support requires the DB2 Client Application Enabler (CAE) Version 5.2, or higher, and MTS must be at Version 2.0 with Hotfix 0772 or later releases.

The installation of the DB2 ODBC driver on Windows 32-bit operating systems will automatically add a new keyword into the registry:

```
HKEY_LOCAL_MACHINE\software\ODBC\odbcinit.ini\IBM DB2 ODBC Driver:
Keyword Value Name: CPTimeout
Data Type: REG_SZ
Value: 60
```

Installation and Configuration: Following is a summary of installation and configuration considerations for MTS. To use DB2's MTS support, the user must:

1. Install MTS and the DB2 client on the same machine where the MTS application runs.
2. If host or AS/400 database servers are to be involved in a multisite update:
   a. Install DB2 Connect Enterprise Edition either on your local machine or on a remote machine. DB2 Connect Enterprise Edition allows host or AS/400 database servers to participate in a multisite update transaction.
   b. Ensure your DB2 Connect Enterprise Edition Server is enabled for multisite update. For information on enabling DB2 Connect for multisite updates please see the DB2 Connect Enterprise Edition Quick Beginnings manual for your platform.

When running DB2 CLI/ODBC applications the following configuration keywords (as set in the `db2cli.ini` file) must not be changed from their default values:

- CONNECTYPE keyword (default 1)
- MULTICONNECT keyword (default 1)
- DISABLEMULTITHREAD keyword (default 1)
- CONNECTIONPOOLING keyword (default 0)
- KEEPCONNECTION keyword (default 0)

DB2 CLI applications written to make use of MTS support must not change the attribute values corresponding to the above keywords. In addition, the application must not change the default values of the following attributes:

- SQL_ATTR_CONNECT_TYPE attribute (default SQL_CONCURRENT_TRANS)
- SQL_ATTR_CONNECTON_POOLING attribute (default SQL_CP_OFF)

Note: Additional technical information may be provided on the IBM web site to assist you with installation and configuration of DB2 MTS support. Set your URL to ″http://www.software.ibm.com/data/db2/library″, and search for a DB2 Universal Database ″Technote″ with the keyword ″MTS″.

Verifying the Installation:

1. Configure DB2 client and DB2 Connect EE to access your DB2 UDB, host, or AS/400 server.
2. Verify the connection from the DB2 CAE machine to the DB2 UDB database servers.
3. Verify the connection from the DB2 Connect machine to your host or AS/400 database server with DB2 CLP and issue a few queries.
4. Verify the connection from the DB2 CAE machine via the DB2 Connect gateway to your host or AS/400 database server and issue a few queries.

Supported DB2 Database Servers: The following servers are supported for multi-site update using MTS-coordinated transactions:

- DB2 Universal Database Enterprise Edition Version 5.2
- DB2 Enterprise - Extended Edition Version 5.2
- DB2 for OS/390
- DB2 for MVS
- DB2 for AS/400
- DB2 for VM&VSE
- DB2 Common Server for SCO, Version 2
- DB2 Universal Database for AIX with PTF U453782
- DB2 Universal Database for HP-UX with PTF U453784
- DB2 Universal Database Enterprise Edition for OS/2 with PTF WR09033
- DB2 Universal Database for SOLARIS with PTF U453783
- DB2 Universal Database Enterprise Edition for Windows NT with PTF WR09034
- DB2 Universal Database Extended Enterprise Edition for UNIX or Windows NT.

MTS Transaction Time-Out and DB2 Connection Behavior: You can set the transaction time-out value in the MTS Explorer tool. Please refer to the on-line *MTS Administrator Guide* for more details.

If a transaction takes longer than the transaction time-out value (default is 60 seconds), MTS will asynchronously issue an abort to all Resource Managers involved, and the whole transaction is aborted.

For the connection to a DB2 server, the abort is translated into a DB2 rollback request. Like any other database requests, the rollback request will be serialized on the connection to guarantee the integrity of the data on the database server.

As a result:

- If the connection is idle, the rollback is executed immediately.
- If a long running SQL statement is being executed, the rollback request will wait until the SQL statement finished before it is executed.

Connection Pooling: Connection pooling enables an application to use a connection from a pool of connections, so that the connection does not need to be reestablished for each use. Once a connection has been created and placed in a pool, an application can reuse that connection without performing a complete connection process. The connection is pooled when the application disconnects from the ODBC data source, and will be given to a new connection whose attributes are the same.

Connection pooling has been a feature of ODBC driver Manager 2.x. With the latest ODBC driver manager (version 3.5) that was shipped with MTS, connection pooling has some configuration changes and new behavior for ODBC connections of transactional MTS COM objects (see "Reusing ODBC Connections Between COM Objects Participating in the Same Transaction" on page 61).

ODBC driver Manager 3.5 requires that the ODBC driver register a new keyword in the registry before it allows connection pooling to be activated. The keyword is:

```
Key Name: SOFTWARE\ODBC\ODBCINST.INI\IBM DB2 ODBC DRIVER
Name: CPTimeout
Type: REG_SZ
Data: 60
```

The DB2 ODBC driver version 6 and later for 32-bit Windows operating system fully supports connection pooling and therefore this keyword is registered. Version 5.2 clients must install Fix Pack 3 (WR09024) or later.

The default value (60) means the connection will be pooled for 60 seconds before it actually is disconnected.

In a busy environment, it is better to increase the CPTimeout value to a large number (Microsoft sometimes suggests 10 minutes for certain environments) to prevent too many physical connects and disconnects, because these consume a large amount of system resources, including system memory and communications stack resource.

Reusing ODBC Connections Between COM Objects Participating in the Same Transaction: ODBC connections in MTS COM objects have connection pooling turned on automatically (whether or not the COM object is transactional) .

For multiple MTS COM objects participating in the same transaction, the connection can be reused between two or more COM objects in the following manner.

Suppose there are two COM objects, COM1 and COM2 that connect to the same ODBC datasource and participate in the same transaction.

After COM1 connects and does its work, it disconnects and the connection is pooled. However, this connection will be reserved for the use of other COM objects of the same transaction. It will be available to other transactions only after the current transaction ends.

When COM2 is invoked in the same transaction, it is given the pooled connection. MTS will ensure that the connection can only be given to the COM objects that are participating in the same transaction.

On the other hand, if COM1 does not explicitly disconnect, then it will tie up the connection until the transaction ends. When COM2 is invoked in the same transaction, a separate connection will be acquired. Subsequently, this transaction ties up two connections instead of one.

This reuse of connection feature for COM objects participating in the same transaction is preferable for the following reasons:

- It uses fewer resources in both the client and the server. Only one connection is needed.
- It eliminates the possibility that two connections participating in the same transaction (accessing the same database server and accessing the same data) can lock one another, because DB2 servers treat different connections from MTS COM objects as separate transactions.

Tuning TCP/IP Communications: If a small CPTimeout value is used in a high-workload environment where too many physical connects and disconnects occur at the same time, the TCP/IP stack may encounter resource problems.

To alleviate this problem, you should use the TCP/IP Registry Entries. These are described in the *Windows NT Resource Guide*, Volume 1. The registry key values are located in HKEY_LOCAL_MACHINE-> SYSTEM-> CurrentControlSet-> Services-> TCPIP-> Parameters.

The default values and suggested settings are as follows:

Name	Default Value	Suggested Value
KeepAlive time	7200000 (2 hours)	Same
KeepAlive interval	1000 (1 second)	10000 (10 seconds)
TcpKeepCnt	120 (2 minutes)	240 (4 minutes)
TcpKeepTries	20 (20 re-tries)	Same
TcpMaxConnectAttempts	3	6
TcpMaxConnectRetransmission	3	6
TcpMaxDataRetransmission	5	8
TcpMaxRetransmissionAttempts	7	10
If the registry value is not defined, then create it.		

Testing DB2 With The MTS ″BANK″ Sample Application: You can use the ″BANK″ sample program that is shipped with MTS to test the setup of the client products and MTS.

Follow these steps:

1. Change the file `\Program Files\Common Files\ODBC\Data Sources\MTSSamples.dsn` so that it looks like this:

   ```
   [ODBC]
   DRIVER=IBM DB2 ODBC DRIVER
   UID=your_user_id
   PWD=your_password
   DSN=your_database_alias
   Description=MTS Samples
   ```

   where:

   - `your_user_id`and `your_password` are the user-ID and password used to connect to the host.
   - `your_database_alias` is the database alias used to connect to the database server.

2. Go to ODBC Administration in the Control Panel, click on System DSN tab and add the data source:
   a. Choose IBM ODBC Driver and click on Finish.
   b. When presented with the list of database aliases, choose the one that was specified previously.
   c. Click on OK
3. Use DB2 CLP to connect to a DB2 database under the ID `your_user_id`, as above.
   a. Bind the db2cli.lst:

```
db2 bind @C:\sqllib\bnd\db2cli.lst blocking all grant public
```

b. Bind the utilities.

   If the server is a DRDA host server, bind `ddcsmvs.lst`, `ddcs400.lst`, or `ddcsvm.lst`, depending on the host that you are connecting to (OS/390, AS/400, or VSE or VM). For example:

   ```
   db2 bind @C:\sqllib\bnd\@ddcsmvs.lst blocking all grant public
   ```

   Otherwise, bind the `db2ubind.lst`:

   ```
   db2 bind @C:\sqllib\bnd\@db2ubind.lst blocking all grant public
   ```

c. Then create the sample table and data for the MTS sample application as follows:

   ```
   DB2 CREATE TABLE ACCOUNT (ACCOUNTNO INT, BALANCE INT)
   DB2 INSERT INTO ACCOUNT VALUES(1, 1)
   ```

4. On the DB2 client, ensure that the database manager configuration parameter *tp_mon_name* is set to ″MTS″.
5. Run the ″BANK″ application. Select the **Account** button and the **Visual C++** option, then submit the request. Other options may use SQL that is specific to SQL Server, and may not work.

Programming Considerations

When running DB2 CLI/ODBC applications the following configuration keywords (as set in the `db2cli.ini` file) must not be changed from their default values:

- CONNECTYPE keyword (default 1)
- MULTICONNECT keyword (default 1)
- DISABLEMULTITHREAD keyword (default 1)
- CONNECTIONPOOLING keyword (default 0)
- KEEPCONNECTION keyword (default 0)

DB2 CLI applications written to make use of MTS support must not change the attribute values corresponding to the above keywords. In addition, the application must not change the values set for the following attributes:

- SQL_ATTR_CONNECT_TYPE attribute (default SQL_CONCURRENT_TRANS)
- SQL_ATTR_CONNECTON_POOLING attribute (default SQL_CP_OFF)

DB2 CLI/ODBC applications written to make use of MTS must use ODBC functions, not CLI functions, to establish a connection.

Restrictions

Mixing Embedded SQL and CLI/ODBC calls in a multisite update environment is supported, but all the same restrictions of writing mixed applications are imposed. Please see “Mixing Embedded SQL and DB2 CLI” on page 136 for more details.

Process-based XA-Compliant Transaction Program Monitor (XA TP)

Process-based XA TPs, such as CICS and Encina, start up one application server per process. In each application-server process, the connections are already established using the XA API (xa_open). This section describes the environment configurations and how to write DB2 CLI/ODBC appliciations to be run under this environment.

Configuration

The XA Transaction Manager must be set up according to the information in the *Administration Guide*.

The setting of the CLI/ODBC configuration keywords for process-based XA TMs are exactly the same as when DB2 is used as the Transaction Manager. Please see “Configuration - DB2 as Transaction Monitor” on page 51 for more inforamtion.

Programming Considerations

DB2 CLI/ODBC applications written for this environment must complete the following steps:

- The application must first call `SQLConnect()` or `SQLDriverConnect()` to associate the TM-opened connections with the CLI/ODBC connection handle. The datasource name (DSN keyword) must be specified. User ID and Password are optional.
- The application must call the XA TM to do commit or rollback. As a result, since the CLI/ODBC driver does not know that the transaction has ended, the application should do the following before exiting:
  - Drop all CLI/ODBC statement handles.
  - Free up the connection handle by calling `SQLDisconnect()` and `SQLFreeConnect()` (or `SQLFreeHandle()`). The actual database connection will not be disconnected until the XA TM performs a xa_close.

Restrictions

Mixing Embedded SQL and CLI/ODBC calls in a multisite update environment is supported, but all the same restrictions of writing mixed applications are imposed. Please see “Mixing Embedded SQL and DB2 CLI” on page 136 for more details.

Host and AS/400 Database Servers

Multisite updates are also supported when a DB2 UDB client is connecting to host or AS/400 DB2 database servers using DB2 Connect.

Configuration

There is no specific DB2 CLI/ODBC client configuration required when connecting to a host or AS/400 database server.

The machine running DB2 Connect may require certain configuration settings to enable running in multisite update mode against the host. For more information please see the *DB2 Connect Quick Beginnings* manual for your platform.

Querying System Catalog Information

Often, one of the first tasks an application performs is to display to the user a list of tables from which one or more tables are selected by the user to work with. Although the application can issue its own queries against the database system catalog to get this type of catalog information, it is best that the application calls the DB2 CLI catalog functions instead. These catalog functions provide a generic interface to issue queries and return consistent result sets across the DB2 family of servers. This allows the application to avoid server specific and release specific catalog queries.

The catalog functions operate by returning to the application a result set through a statement handle. Calling these functions is conceptually equivalent to using `SQLExecDirect()` to execute a select against the system catalog tables. After calling these functions, the application can fetch individual rows of the result set as it would process column data from an ordinary `SQLFetch()`. The DB2 CLI catalog functions are:

- "SQLColumnPrivileges - Get Privileges Associated With The Columns of A Table" on page 310
- "SQLColumns - Get Column Information for a Table" on page 315
- "SQLForeignKeys - Get the List of Foreign Key Columns" on page 420
- "SQLPrimaryKeys - Get Primary Key Columns of A Table" on page 590
- "SQLProcedureColumns - Get Input/Output Parameter Information for A Procedure" on page 594
- "SQLProcedures - Get List of Procedure Names" on page 603
- "SQLSpecialColumns - Get Special (Row Identifier) Columns" on page 726

- "SQLStatistics - Get Index and Statistics Information For A Base Table" on page 733
- "SQLTablePrivileges - Get Privileges Associated With A Table" on page 740
- "SQLTables - Get Table Information" on page 745
- "SQLGetTypeInfo - Get Data Type Information" on page 555

The result sets returned by these functions are defined in the descriptions for each catalog function. The columns are defined in a specified order. In future releases, other columns may be added to the end of each defined result set, therefore applications should be written in a way that would not be affected by such changes.

Some of the catalog functions result in execution of fairly complex queries, and for this reason should only be called when needed. It is recommended that the application save the information returned rather than making repeated calls to get the same information.

Input Arguments on Catalog Functions

All of the catalog functions have *CatalogName* and *SchemaName* (and their associated lengths) on their input argument list. Other input arguments may also include *TableName*, *ProcedureName*, or *ColumnName* (and their associated lengths). These input arguments are used to either identify or constrain the amount of information to be returned. *CatalogName*, however, must always be a null pointer (with its length set to 0) as DB2 CLI does not support three-part naming.

In the Function Arguments sections for these catalog functions in "Chapter 5. DB2 CLI Functions" on page 209, each of the above input arguments are described either as a pattern-value or just as an ordinary argument. For example, `SQLColumnPrivileges()` treats *SchemaName* and *TableName* as ordinary arguments and *ColumnName* as a pattern-value.

Inputs treated as ordinary arguments are taken literally and the case of letters is significant. The argument does not qualify a query but rather identifies the information desired. An error results if the application passes a null pointer for this argument.

Inputs treated as pattern-values are used to constrain the size of the result set by including only matching rows as though the underlying query were qualified by a WHERE clause. If the application passes a null pointer for a pattern-value input, the argument is not used to restrict the result set (that is, there is no WHERE clause). If a catalog function has more than one pattern-value input argument, they are treated as though the WHERE clauses

in the underlying query were joined by AND; a row appears in this result set only if it meets all the conditions of the WHERE clauses.

Each pattern-value argument can contain:

- The underscore (_) character which stands for any single character.
- The percent (%) character which stands for any sequence of zero or more characters. Note that providing a pattern-value containing a single % is equivalent to passing a null pointer for that argument.
- Characters which stand for themselves. The case of a letter is significant.

These argument values are used on conceptual LIKE predicate(s) in the WHERE clause. To treat the metadata characters (_, %) as themselves, an escape character must immediately precede the _ or %. The escape character itself can be specified as part of the pattern by including it twice in succession. An application can determine the escape character by calling `SQLGetInfo()` with SQL_SEARCH_PATTERN_ESCAPE.

Catalog Functions Example

In the `browser.c` sample application:

- A list of all tables are displayed for the specified schema (qualifier) name or search pattern.
- Column, special column, foreign key, and statistics information is returned for a selected table.

Output from the `browser.c` sample is shown below, relevant segments of the sample are listed for each of the catalog functions.

```
Enter Search Pattern for Table Schema Name:
STUDENT
Enter Search Pattern for Table Name:
%
### TABLE SCHEMA              TABLE_NAME                TABLE_TYPE
    ------------------------ ------------------------ ----------
1   STUDENT                  CUSTOMER                  TABLE
2   STUDENT                  DEPARTMENT                TABLE
3   STUDENT                  EMP_ACT                   TABLE
4   STUDENT                  EMP_PHOTO                 TABLE
5   STUDENT                  EMP_RESUME                TABLE
6   STUDENT                  EMPLOYEE                  TABLE
7   STUDENT                  NAMEID                    TABLE
8   STUDENT                  ORD_CUST                  TABLE
9   STUDENT                  ORD_LINE                  TABLE
10  STUDENT                  ORG                       TABLE
11  STUDENT                  PROD_PARTS                TABLE
12  STUDENT                  PRODUCT                   TABLE
13  STUDENT                  PROJECT                   TABLE
14  STUDENT                  STAFF                     TABLE
Enter a table Number and an action:(n [Q | C | P | I | F | T |O | L])
```

```
|Q=Quit     C=cols     P=Primary Key I=Index   F=Foreign Key |
|T=Tab Priv O=Col Priv S=Stats       L=List Tables           |
1c
Schema: STUDENT  Table Name: CUSTOMER
   CUST_NUM, NOT NULLABLE, INTeger (10)
   FIRST_NAME, NOT NULLABLE, CHARacter (30)
   LAST_NAME, NOT NULLABLE, CHARacter (30)
   STREET, NULLABLE, CHARacter (128)
   CITY, NULLABLE, CHARacter (30)
   PROV_STATE, NULLABLE, CHARacter (30)
   PZ_CODE, NULLABLE, CHARacter (9)
   COUNTRY, NULLABLE, CHARacter (30)
   PHONE_NUM, NULLABLE, CHARacter (20)
>> Hit Enter to Continue<<

1p
Primary Keys for STUDENT.CUSTOMER
 1  Column: CUST_NUM            Primary Key Name: = NULL
>> Hit Enter to Continue<<

1f
Primary Key and Foreign Keys for STUDENT.CUSTOMER
  CUST_NUM  STUDENT.ORD_CUST.CUST_NUM
      Update Rule SET NULL , Delete Rule: NO ACTION
>> Hit Enter to Continue<<
```

Scrollable Cursors

DB2 CLI supports Scrollable Cursors; the ability to scroll through a cursor:

- 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
- From a previously stored location in the cursor

There are two types of scrollable cursors supported by DB2 CLI:

- “Static, Read-Only Cursor”
- “Keyset-Driven Cursor” on page 69

After a description of each cursor type there is the section “Deciding on Which Cursor Type to Use” on page 70.

Static, Read-Only Cursor

This type of scrollable cursor is static; once it is created no rows will be added or removed, and no values in any rows will change. The cursor is not affected by other applications accessing the same data.

The cursor is also read-only. It is not possible for the application to change any values. How the rows of the cursor are locked, if at all, is determined by the isolation level of the statement used to create the cursor. Refer to the *SQL Reference* for a complete discussion of isolation levels and their effect.

Keyset-Driven Cursor

This type of scrollable cursor adds two features that static cursors do not have; the ability to detect changes to the underlying data, and the ability to use the cursor to make changes to the underlying data.

When a keyset-driven cursor is first opened, it stores the keys in a keyset for the life of the entire result set. This is used to determine the order and set of rows which are included in the cursor. As the cursor scrolls through the result set, it uses the keys in this keyset to retrieve the current data values for each row. Each time the cursor refetches the row it retrieves the most recent values in the datbase, not the values that existed when the cursor is first opened. For this reason no changes will be reflected in a row until the application scrolls past the row.

There are various types of changes to the underlying data that a keyset-driven cursor may or may not reflect:

Changed values in existing rows
: The cursor will relect these types of changes. Because the cursor refetches the row from the database each time it is required, keyset-driven cursors always detect changes made by themselves and others.

Deleted rows
: The cursor will also relect these types of changes. If a row in the rowset is deleted after the keyset is generated, it will appear as a “hole” in the cursor. When the cursor goes to refetch the row from the database it will realize that it is no longer there.

Added rows
: The cursor will NOT relect these types of changes. The set of rows is determined once, when the cursor is first opened. It does not re-issue the select statement to see if new rows have been added that should be included.

Keyset-driven cursors can also be used to modify the rows in the result set with calls to either `SQLBulkOperations()` or `SQLSetPos()`.

Using SQLBulkOperations() with Keyset-driven Cursors

`SQLBulkOperations()` can be used to add, update, delete, or fetch a set of rows. The *Operation* argument is used to indicate how the rows are to be

updated. For more information see one of the following sections in “SQLBulkOperations - Add, Update, Delete or Fetch a Set of Rows” on page 276:

- “Performing Bulk Inserts” on page 277
- “Performing Bulk Updates Using Bookmarks” on page 278
- “Performing Bulk Fetches Using Bookmarks” on page 279
- “Performing Bulk Deletes Using Bookmarks” on page 280

Using SQLSetPos() with Keyset-driven Cursors

`SQLSetPos()` can be used to update or delete a set of rows. The *Operation* argument is used to indicate how the rows are to be updated. For more information see one of the following sections in “SQLSetPos - Set the Cursor Position in a Rowset” on page 691:

- “SQL_UPDATE Option” on page 694
- “SQL_DELETE Option” on page 694

Deciding on Which Cursor Type to Use

The first decision to make is between a static cursor and a scrollable cursor. If your application does not need the additional features of a scrollable cursor then a static cursor should be used.

If a scrollable cursor is required then you have to decide between a static or keyset-driven cursor. A static cursor involves the least overhead. If the application does not need the additional features of a keyset-driven cursor then a static cursor should be used.

If the application needs to detect changes to the underlying data, or needs to add, update, or delete data from the cursor then it must use a keyset-driven cursor.

To determine the types of cursors supported by the driver and DBMS the application should call `SQLGetInfo()` with an *InfoType* of:

- SQL_DYNAMIC_CURSOR_ATTRIBUTES1
- SQL_DYNAMIC_CURSOR_ATTRIBUTES2
- SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1
- SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2
- SQL_KEYSET_CURSOR_ATTRIBUTES1
- SQL_KEYSET_CURSOR_ATTRIBUTES2
- SQL_STATIC_CURSOR_ATTRIBUTES1

- SQL_STATIC_CURSOR_ATTRIBUTES2

Specifying the Rowset Returned from the Result Set

It is important to understand the following terms:

result set The complete set of rows that are generated by the SQL SELECT statement. Once created the result set will not change.

rowset The subset of rows from the result set that is returned after each fetch. The application indicates the size of the rowset before the first fetch of data, and can modify the size before each subsequent fetch. Each call to `SQLFetchScroll()` populates the rowset with the appropriate rows from the result set.

bookmark It is possible to store a pointer to a specific row in the result set; a bookmark. Once stored, the application can continue to move throughout the result set, then return to the bookmarked row to generate a rowset. See "Using Bookmarks with Scrollable Cursors" on page 78 for complete details.

The position of the rowset within the result set is specified in the call to `SQLFetchScroll()`. For example, the following call would generate a rowset starting on the 11th row in the result set (step 5 in Figure 7 on page 72):

```
SQLFetchScroll(hstmt,   /* Statement handle */
    SQL_FETCH_ABSOLUTE, /* FetchOrientation value */
    11);                /* Offset value */
```

Scroll bar operations of a screen-based application can be mapped directly to the positioning of a rowset. By setting the rowset size to the number of lines displayed on the screen, the application can map the movement of the scroll bar to calls to `SQLFetchScroll()`.

Rowset Retrieved	FetchOrientation Value	Scroll bar
First rowset	SQL_FETCH_FIRST	Home: Scroll bar at the top
Last rowset	SQL_FETCH_LAST	End: Scroll bar at the bottom
Next rowset	SQL_FETCH_NEXT (same as calling `SQLFetch()`)	Page Down
Previous rowset	SQL_FETCH_PRIOR	Page Up
Rowset starting on next row	SQL_FETCH_RELATIVE with *FetchOffset* set to 1	Line Down

Rowset Retrieved	FetchOrientation Value	Scroll bar
Rowset starting on previous row	SQL_FETCH_RELATIVE with *FetchOffset* set to -1	Line Up
Rowset starting on a specific row	SQL_FETCH_ABSOLUTE with *FetchOffset* set to an offset from the start (a positive value) or the end (a negative value) of the result set	Application generated
Rowset starting on a previously bookmarked row	SQL_FETCH_BOOKMARK with *FetchOffset* set to a positive or negative offset from the bookmarked row (see "Using Bookmarks with Scrollable Cursors" on page 78 for more information)	Application generated

The following figure demonstrates a number of calls to `SQLFetchScroll()` using various *FetchOrientation* values. The result set includes all of the rows (from 1 to n), and the rowset size is 3. The order of the calls is indicated on the left, and the *FetchOrientation* values are indicated on the right.

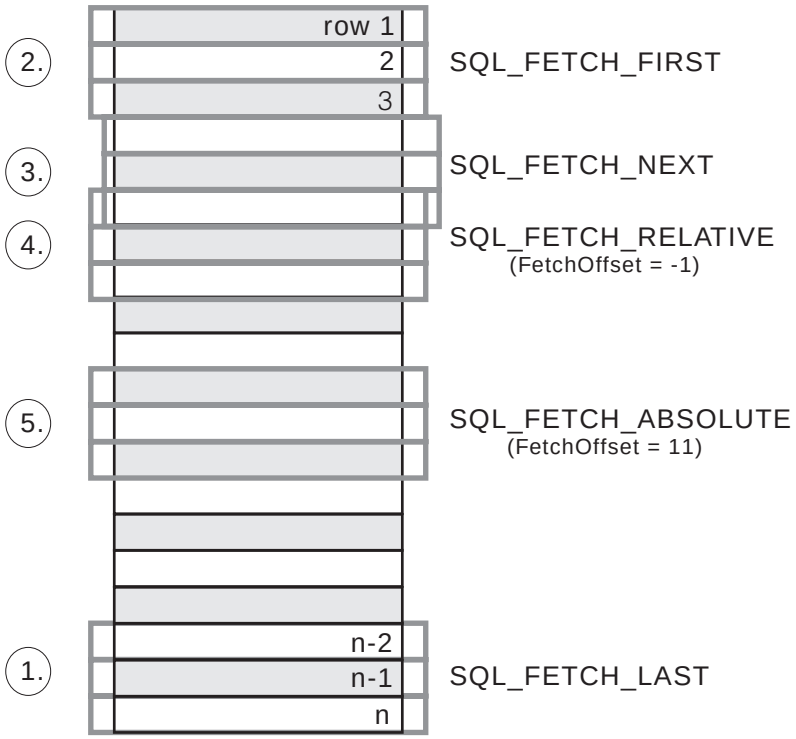


Figure 7. Example of Retrieving Rowsets

For more details see “Cursor Positioning Rules” on page 409 in SQLFetchScroll().

Size of Returned Rowset

The statement attribute SQL_ATTR_ROW_ARRAY_SIZE is used to declare the number of rows in the rowset. For example, to declare a rowset size of 35 rows, the following call would be used:

```
/* CLI Sample: sfetch.c */
/*...*/
#define ROWSET_SIZE 35
/*...*/
    rc = SQLSetStmtAttr(
                hstmt,
                SQL_ATTR_ROW_ARRAY_SIZE,
                (SQLPOINTER) ROWSET_SIZE,
                0);
```

The application cannot assume that the entire rowset will contain data. It must check the rowset size after each rowset is created because there are instances where the rowset will not contain a complete set of rows. For instance, consider the case where the rowset size is set to 10, and SQLFetchScroll() is called using SQL_FETCH_ABSOLUTE and *FetchOffset* set to -3. This will attempt to return 10 rows starting 3 rows from the end of the result set. Only the first three rows of the rowset will contain meaningful data, however, and the application must ignore the rest of the rows.

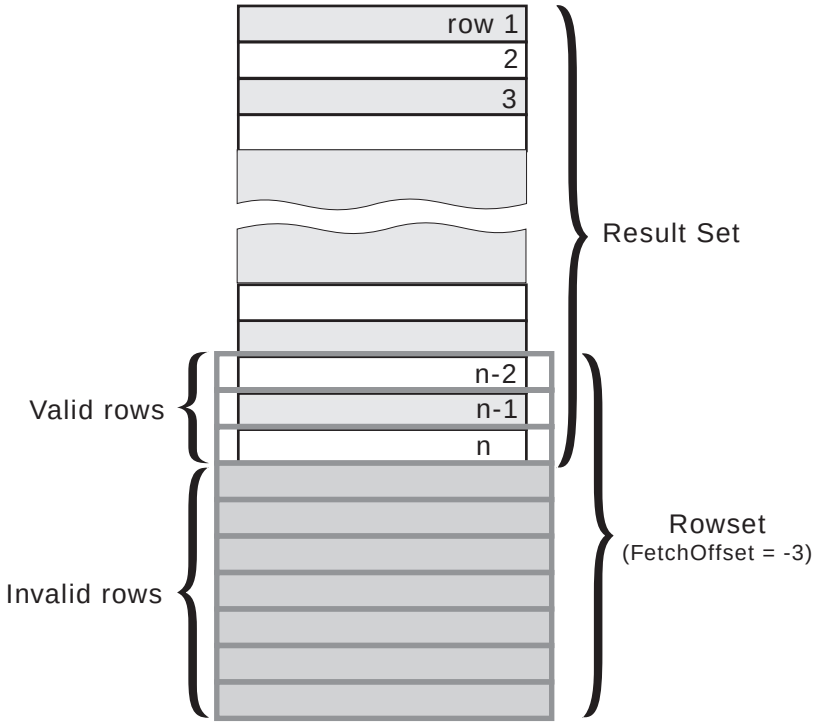


Figure 8. Partial Rowset Example

See “Setting the Rowset size” on page 75 for more information on using the statement attribute SQL_ATTR_ROW_ARRAY_SIZE.

Row Status Array

The row status array provides additional information about each row in the rowset. After each call to SQLFetchScroll() the array is updated. The application must declare an array (of type SQLUSMALLINT) with the same number of rows as the size of the rowset (the statement attribute SQL_ATTR_ROW_ARRAY_SIZE). The address of this array is then specified with the statement attribute SQL_ATTR_ROW_STATUS_PTR.

```
/* CLI Sample: sfetch.c */
/* ... */
SQLUSMALLINT    row_statusÁROWSET_SIZE‖;
/* ... */
/* Set a pointer to the array to use for the row status */
rc = SQLSetStmtAttr(
            hstmt,
            SQL_ATTR_ROW_STATUS_PTR,
            (SQLPOINTER) row_status,
            0);
/* ... */
```

If the call to SQLFetchScroll() does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO the the contents of the row status buffer is undefined, otherwise the following values are returned:

Row status array value	Description
SQL_ROW_SUCCESS	The row was successfully fetched.
SQL_ROW_SUCCESS_WITH_INFO	The row was successfully fetched. However, a warning was returned about the row.
SQL_ROW_ERROR	An error occurred while fetching the row.
SQL_ROW_NOROW	The rowset overlapped the end of the result set and no row was returned that corresponded to this element of the row status array.
SQL_ROW_ADDED	The row was inserted by SQLBulkOperations(). If the row is fetched again, or is refreshed by SQLSetPos() its status is SQL_ROW_SUCCESS. This value is not set by SQLFetch() or SQLFetchScroll().
SQL_ROW_UPDATED	The row was successfully fetched and has changed since it was last fetched from this result set. If the row is fetched again from this result set, or is refreshed by SQLSetPos(), the status changes to the row's new status.
SQL_ROW_DELETED	The row has been deleted since it was last fetched from this result set.

In Figure 8 on page 74, the first 3 rows of the row status array would contain the value SQL_ROW_SUCCESS; the remaining 7 rows would contain SQL_ROW_NOROW.

Typical Scrollable Cursors Application

Each application that will make use of scrollable cursors must complete the following steps, in the following order:

1. Set Up the Environment

The following additional statement attributes are required when using scrollable cursors in DB2 CLI applications. See "SQLSetStmtAttr - Set Options Related to a Statement" on page 702 for complete details.

Setting the Rowset size

Set the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that you want returned from each call to SQLFetchScroll().

The default value is 1.

Type of scrollable cursor

DB2 CLI supports either static, read-only cursors, or keyset-driven cursors. Use SQLSetStmtAttr() to set the SQL_ATTR_CURSOR_TYPE statement attribute to either SQL_CURSOR_STATIC or SQL_CURSOR_KEYSET_DRIVEN. ODBC defines other scrollable cursors types, but they cannot be used with DB2 CLI.

This value must be set or the default value of SQL_CURSOR_FORWARD_ONLY will be used.

Location to store number of rows returned

The application needs a way to determine how many rows were returned in the rowset from each call to SQLFetchScroll(). The number of rows returned in the rowset can at times be less than the maximum size of the rowset which was set using SQL_ATTR_ROW_ARRAY_SIZE.

Set the SQL_ATTR_ROWS_FETCHED_PTR statement attribute as a pointer to a SQLUINTEGER variable. This variable will then contain the number of rows returned in the rowset after each call to SQLFetchScroll().

Array to use for the row status

Set the SQL_ATTR_ROW_STATUS_PTR statement attribute as a pointer to the SQLUSMALLINT array that is used to store the row status. This array will then be updated after each call to SQLFetchScroll().

For more information see “Row Status Array” on page 74.

Will bookmarks be used?

If you plan on using bookmarks in your scrollable cursor then you must set the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE.

The following example demonstrates the required calls to SQLSetStmtAttr():

```
/* CLI Sample: sfetch.c */
/* ... */

    /* Set the number of rows in the rowset */
    rc = SQLSetStmtAttr(
                hstmt,
                SQL_ATTR_ROW_ARRAY_SIZE,
                (SQLPOINTER) ROWSET_SIZE,
```

```
            0);
    CHECK_STMT(hstmt, rc);

    /* Set the SQL_ATTR_ROWS_FETCHED_PTR statement attribute to */
    /* point to the variable numrowsfetched: */
    rc = SQLSetStmtAttr(
            hstmt,
            SQL_ATTR_ROWS_FETCHED_PTR,
            &numrowsfetched,
            0);
    CHECK_STMT(hstmt, rc);

    /* Set a pointer to the array to use for the row status */
    rc = SQLSetStmtAttr(
            hstmt,
            SQL_ATTR_ROW_STATUS_PTR,
            (SQLPOINTER) row_status,
            0);
    CHECK_STMT(hstmt, rc);

    /* Set the cursor type */
    rc = SQLSetStmtAttr(
            hstmt,
            SQL_ATTR_CURSOR_TYPE,
            (SQLPOINTER) SQL_CURSOR_STATIC,
            0);
    CHECK_STMT(hstmt, rc);

    /* Indicate that we will use bookmarks by setting the */
    /* SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE: */
    rc = SQLSetStmtAttr(
            hstmt,
            SQL_ATTR_USE_BOOKMARKS,
            (SQLPOINTER) SQL_UB_VARIABLE,
            0);
    CHECK_STMT(hstmt, rc);

/* ... */
```

2. Execute SQL SELECT Statement and Bind the Results

Follow the usual DB2 CLI process for executing an SQL statement and binding the result set. The application can call `SQLRowCount()` to determine the number of rows in the overall result set. Scrollable cursors support the use of both column wise and row wise binding. The CLI sample program sfetch.c demonstrates the use of both methods.

3. Fetch a Rowset of Rows at a time from the Result Set

At this point the application can read information from the result set using the following steps:

1. Use SQLFetchScroll() to fetch a rowset of data from the result set. The *FetchOrientation* argument is used to indicate the location of the rowset in the result set. See "Specifying the Rowset Returned from the Result Set" on page 71 for more details.

   A typical call to SQLFetchScroll() to retrieve the first rowset of data would be as follows:

   ```
   SQLFetchScroll(hstmt, SQL_FETCH_FIRST, 0);
   ```

2. Calculate the number of rows returned in the result set. This value is set automatically after each call to SQLFetchScroll(). In the example above we set the statement attribute SQL_ATTR_ROWS_FETCHED_PTR to the variable numrowsfetched which will therefore contain the number of rows fetched after each SQLFetchScroll() call.

   If you have set the SQL_ATTR_ROW_STATUS_PTR statement attribute then the row status array will also be updated for each possible row in the rowset. For more information see "Row Status Array" on page 74.

3. Display or manipulate the data in the rows returned.

4. Free the Statement which then Closes the Result Set

Once the application has finished retrieving information it should follow the usual DB2 CLI process for freeing a statement handle.

Using Bookmarks with Scrollable Cursors

You can save a pointer to any row in the result set; a bookmark. The application can then use that bookmark as a relative position to retrieve a rowset of information. You can retrieve a rowset starting from the bookmarked row, or specify a positive or negative offset.

Once you have positioned the cursor to a row in a rowset using SQLSetPos(), you can obtain the bookmark value from column 0 using SQLGetData(). In most cases you will not want to bind column 0 and retrieve the bookmark value for every row, but use SQLGetData() to retrieve the bookmark value for the specific row you require.

A bookmark is only valid within the result set in which it was created. The bookmark value will be different if you select the same row from the same result set in two different cursors.

The only valid comparison is a byte-by-byte comparison between two bookmark values obtained from the same result set. If they are the same then they both point to the same row. Any other mathematical calculations or comparisons between bookmarks will not provide any useful information. This includes comparing bookmark values within a result set, and between result sets.

Typical Bookmark Usage

To make use of bookmarks the following steps must be followed in addition to the steps described in “Typical Scrollable Cursors Application” on page 75.

Set up the Environment: To use bookmarks you must set the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE. This is in addition to the other statement attributes required for scrollable cursors.

ODBC defines both variable and fixed-length bookmarks. DB2 CLI only supports the newer, variable-length bookmarks.

Get the Bookmark Value from the Desired Row in a Rowset: The application must execute the SQL SELECT statement and use `SQLFetchScroll()` to retrieve a rowset with the desired row. `SQLSetPos()` is then used to position the cursor within the rowset. Finally, the bookmark value is obtained from column 0 using `SQLGetData()` and stored in a variable.

Set the Bookmark Value Statement Attribute: The statement attribute SQL_ATTR_FETCH_BOOKMARK_PTR is used to store the location for the next call to `SQLFetchScroll()` that uses a bookmark.

Once you have the bookmark value using `SQLGetData()` (the variable *abookmark* below), call `SQLSetStmtAttr()` as follows:

```
rc = SQLSetStmtAttr(
           hstmt,
           SQL_ATTR_FETCH_BOOKMARK_PTR,
           (SQLPOINTER) abookmark,
           0);
```

Retrieve a Rowset Based on the Bookmark: Once the bookmark value is stored, the application can continue to use `SQLFetchScroll()` to retrieve data from the result set.

The application can then move throughout the result set, but still retrieve a rowset based on the location of the bookmarked row at any point before the cursor is closed.

The following call to `SQLFetchScroll()` will retrieve a rowset starting with the bookmarked row:

```
rc = SQLFetchScroll(hstmt, SQL_FETCH_BOOKMARK, 0);
```

The value 0 specifies the offset. You would specify -3 to begin the rowset 3 rows before the bookmarked row, or specify 4 to begin 4 rows after.

Note that the variable used to store the bookmark value is not specified in the SQLFetchScroll() call. It was set in the previous step using the statement attribute SQL_ATTR_FETCH_BOOKMARK_PTR.

Sending/Retrieving Long Data in Pieces

When manipulating long data, it may not be feasible for the application to load the entire parameter data value into storage at the time the statement is executed, or when the data is fetched from the database. A method has been provided to allow the application to handle the data in a piecemeal fashion. The technique to send long data in pieces is called *Specifying Parameter Values at Execute Time* because it can also be used to specify values for fixed size non-character data types such as integers.

An application can also use the SQLGetSubString() function to retrieve a portion of a large object value. See Figure 16 on page 119 in "Using Large Objects" on page 116 for details.

Specifying Parameter Values at Execute Time

A bound parameter for which value is prompted at execution time instead of stored in memory before calling SQLExecute() or SQLExecDirect() is called a data-at-execute parameter. To indicate such a parameter on an SQLBindParameter() call, the application:

- Sets the input data length pointer to point to a variable that, at execute time, will contain the value SQL_DATA_AT_EXEC.
- If there is more than one data-at-execute parameter, sets each input data pointer argument to some value that it will recognize as uniquely identifying the field in question.

If there are any data-at-execute parameters when the application calls SQLExecDirect() or SQLExecute(), the call returns with SQL_NEED_DATA to prompt the application to supply values for these parameters. The application responds as follows:

1. It calls SQLParamData() to conceptually advance to the first such parameter. SQLParamData() returns SQL_NEED_DATA and provides the contents of the input data pointer argument specified on the associated SQLBindParameter() call to help identify the information required.
2. It calls SQLPutData() to pass the actual data for the parameter. Long data can be sent in pieces by calling SQLPutData() repeatedly.
3. It calls SQLParamData() again after it has provided the entire data for this data-at-execute parameter. If more data-at-execute parameters exist, SQLParamData() again returns SQL_NEED_DATA and the application repeats steps 2 and 3 above.

When all data-at-execute parameters have been assigned values, SQLParamData() completes execution of the SQL statement and produces a return value and diagnostics as the original SQLExecDirect() or SQLExecute() would have produced. The right side of Figure 9 on page 82 illustrates this flow.

While the data-at-execution flow is in progress, the only DB2 CLI functions the application can call are:

- SQLParamData() and SQLPutData() as given in the sequence above.
- The SQLCancel() function which is used to cancel the flow and force an exit from the loop(s) on the right side of Figure 9 on page 82 without executing the SQL statement.
- The SQLGetDiagRec() function. The application also must not end the transaction nor set any connection attributes.

Using the parameter at execute time technique to input Large Object data may require the creation and use of a temporary file at the client. For alternative methods to input long data, refer to "Using Large Objects" on page 116.

Fetching Data in Pieces

Typically, based on its knowledge of a column in the result set (via SQLDescribeCol() or prior knowledge), the application may choose to allocate the maximum memory the column value could occupy and bind it via SQLBindCol(). However, in the case of character and binary data, the column can be arbitrarily long. If the length of the column value exceeds the length of the buffer the application can allocate or afford to allocate, a feature of SQLGetData() lets the application use repeated calls to obtain in sequence the value of a single column in more manageable pieces.

Basically, as shown on the left side of Figure 9 on page 82, a call to SQLGetData() returns SQL_SUCCESS_WITH_INFO (with SQLSTATE 01004) to indicate more data exists for this column. SQLGetData() is called repeatedly to get the remaining pieces of data until it returns SQL_SUCCESS, signifying that the entire data have been retrieved for this column.

The function SQLGetSubString() can also be used to retrieve a specific portion of a large object value. See "SQLGetSubString - Retrieve Portion of A String Value" on page 550 for more information. For other alternative methods to retrieve long data, refer to "Using Large Objects" on page 116.

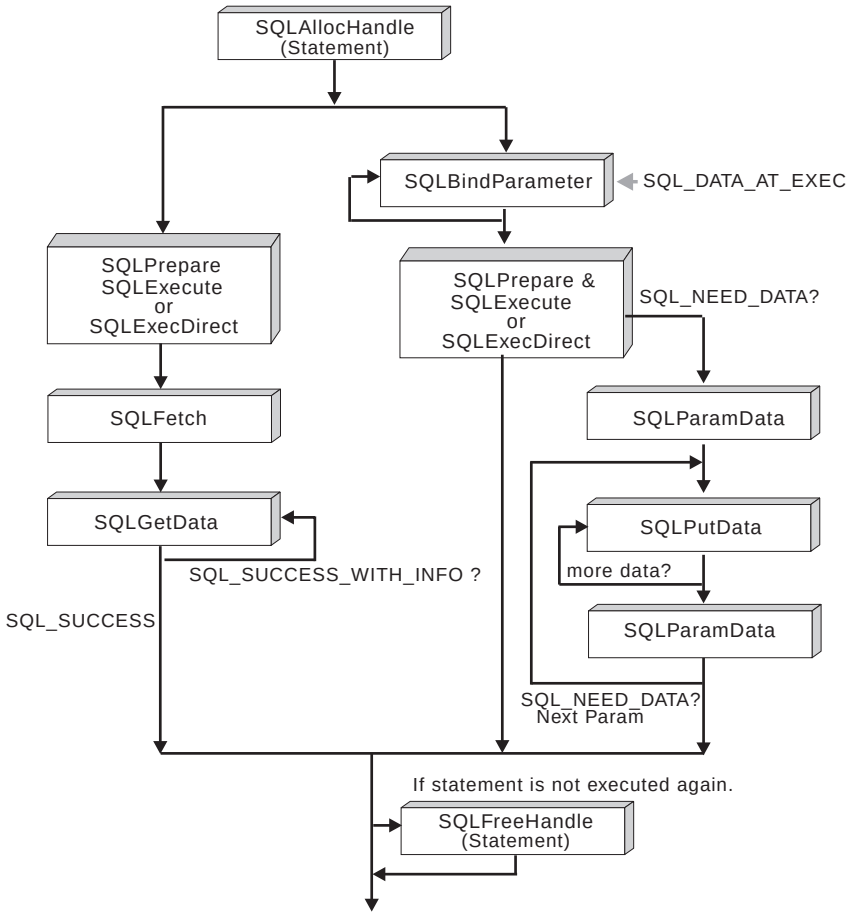


Figure 9. Piecewise Input and Retrieval

Piecewise Input and Retrieval Example

For an example of piecewise input of an image blob refer to `picin2.c`, shown in "Example" on page 612.

For an example of piecewise retrieval of an image blob refer to `showpic2.c`, shown in "Example" on page 454.

Using Arrays to Input Parameter Values

For some data entry and update applications (especially graphical), users may often insert, delete, or change many cells in a data entry form and then ask for the data to be sent to the database. For these situations of bulk insert, delete, or update, DB2 CLI provides an array input method to save the application from having to call SQLExecute() repeatedly on the same INSERT, DELETE, or UPDATE statement. In addition, there is significant savings in network flows.

There are two ways an application can bind the parameter markers in an SQL statement to arrays:

- *Column-Wise Array Insert* (uses column-wise binding): A different array is bound to each parameter.
- *Row-Wise Array Insert* (uses row-wise binding): A structure is created to store a complete set of parameters for a statement. An array of these structures is created and bound to the parameters. Parameter binding offsets (described in the next section) can only be used with row-wise bindings.

SQLBindParameter() is still used to bind buffers to parameters, the only difference is that the addresses passed are array addresses, not single-variable addresses. The application must also set the SQL_ATTR_PARAM_BIND_TYPE statement attribute to specify whether column-wise or row-wise binding will be used.

Column-Wise Array Insert

This method involves the binding of parameter marker(s) to array(s) of storage locations via the SQLBindParameter() call. For character and binary input data, the application uses the maximum input buffer size argument (*BufferLength*) on SQLBindParameter() call to indicate to DB2 CLI the location of values in the input array. For other input data types, the length of each element in the array is assumed to be the size of the C data type. The statement attribute SQL_ATTR_PARAMSET_SIZE must be set (with a call to SQLSetStmtAttr()) to the size of the array before the execution of the SQL statement.

Suppose for Figure 10 on page 84 there is an application that allows the user to change values in the OVERTIME_WORKED and OVERTIME_PAID columns of a time sheet data entry form. Also suppose that the primary key of the underlying EMPLOYEE table is EMPLOY_ID. The application can then request to prepare the following SQL statement:

```
UPDATE EMPLOYEE SET OVERTIME_WORKED= ? and OVERTIME_PAID= ?
WHERE EMPLOY_ID=?
```

When the user has entered all the changes, the application counts that n rows are to change and allocates m=3 arrays to store the changed data and the primary key. Then it calls SQLBindParameter() to bind the three parameter markers to the location of three arrays in memory. Next it sets the statement attribute SQL_ATTR_PARAMSET_SIZE (with a call to SQLSetStmtAttr()) to specify the number of rows to change (the size of the array). Then it calls SQLExecute() once and all the updates are sent to the database. This is the flow shown on the right side of Figure 10.

The basic method is shown on the left side of Figure 10 where SQLBindParameter() is called to bind the three parameter markers to the location of three variables in memory. SQLExecute() is called to send the first set of changes to the database. The variables are updated to reflect values for the next row of changes and again SQLExecute() is called. Note that this method has n-1 extra SQLExecute() calls.

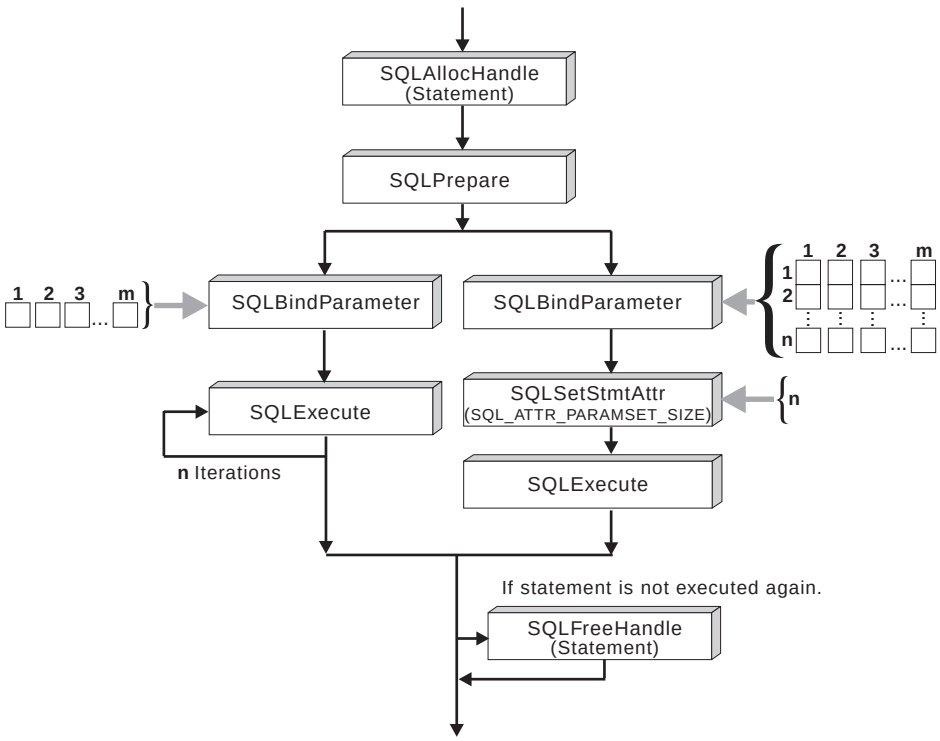


Figure 10. Column-Wise Array Insert

See “Retrieving Diagnostic Information” on page 86 for information on errors that can be accessed by the application.

Row-Wise Array Insert

The first step, when using row-wise array insert, is to create a structure that contains two elements for each parameter. The first element for each parameter holds the length/indicator buffer, and the second element holds the value itself. Once the structure is defined the application must allocate an array of these structures. The number of rows in the array corresponds to the number of values that will be used for each parameter.

```
struct { SQLINTEGER La; SQLINTEGER A;  /* Information for parameter A */
         SQLINTEGER Lb; SQLCHAR B[4];  /* Information for parameter B */
         SQLINTEGER Lc; SQLCHAR C[11]; /* Information for parameter C */
       } R[n];
```

Figure 11 on page 86 shows the structure R with three parameters, in an array of *n* rows. The array can then be populated with the appropriate data.

Once the array is created and populated the application must indicate that row-wise binding is going to be used. It does this by setting the statement attribute SQL_ATTR_PARAM_BIND_TYPE to the length of the structure created. The statement attribute SQL_ATTR_PARAMSET_SIZE must also be set to the number of rows in the array.

Each parameter can now be bound to the appropriate two elements of the structure (in the first row of the array) using SQLBindParameter().

```
/* Parameter A */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
         5, 0, &R[0].A, 0, &R.La);

/* Parameter B */
rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
         10, 0, R[0].B, 10, &R.Lb);

/* Parameter C */
rc = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
         3, 0, R[0].C, 3, &R.Lc);
```

At this point the application can call SQLExecute() once and all of the updates are sent to the database.

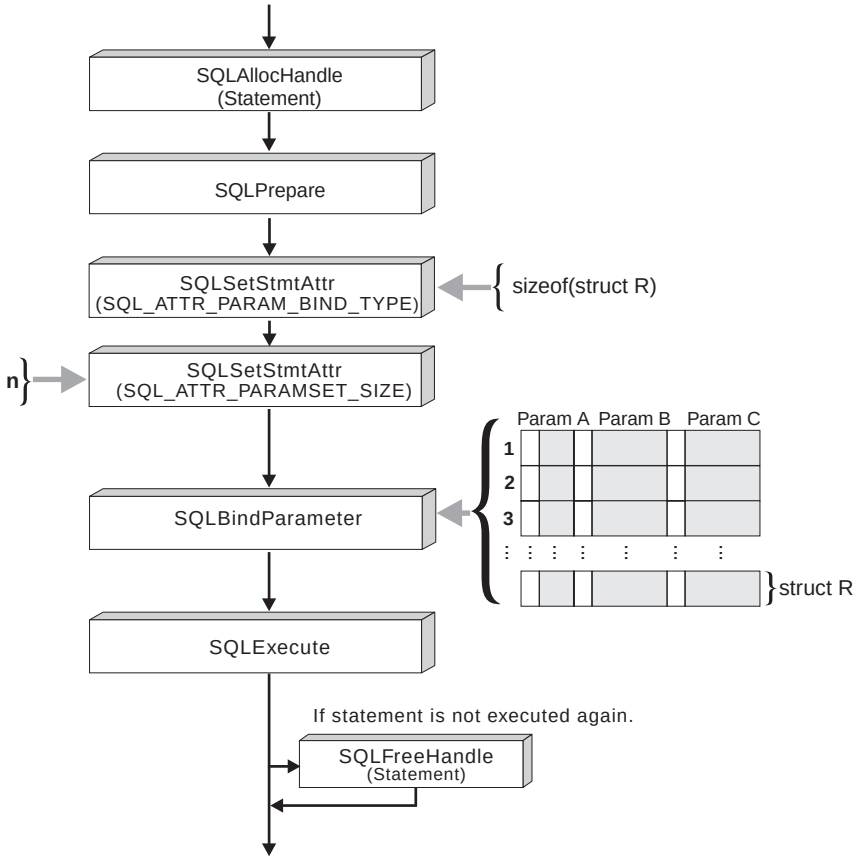


Figure 11. Row-Wise Array Insert

See “Retrieving Diagnostic Information” for information on errors that can be accessed by the application.

Retrieving Diagnostic Information

A *parameter status array* can be populated after the `SQLExecute()` or `SQLExecDirect()` call. The array contains information about the processing of each set of parameters. See the statement attribute SQL_ATTR_PARAM_STATUS_PTR, or the corresponding IPD descriptor header field SQL_DESC_ARRAY_STATUS_PTR, for complete details.

The statement attribute SQL_ATTR_PARAMS_PROCESSED, or the corresponding IPD descriptor header field SQL_DESC_ROWS_PROCESSED_PTR, can be used to return the number of sets of parameters that have been processed. See these attributes in the description of `SQLSetStmtAttr()` or `SQLSetDescField()`.

Once the application has determined what parameters had errors, it can use the statement attribute SQL_ATTR_PARAM_OPERATION_PTR, or the corresponding APD descriptor header field SQL_DESC_ARRAY_STATUS_PTR, (both of which point to an array of values) to control which sets of parameters are ignored in a second call to SQLExecute() or SQLExecDirect(). See these attributes in the description of SQLSetStmtAttr() or SQLSetDescField().

Other Information

In environments where the underlying support allows Compound SQL (DB2 Universal Database, or DRDA environments with DB2 Connect V 2.3 or higher), there is additional savings in network flow. All the data in the array(s) together with the execute request are packaged together as one flow. For DRDA environments, the underlying Compound SQL support is always NOT ATOMIC COMPOUND SQL. This means that execution will continue even if an error is detected with one of the intermediate array elements. When SQLRowCount() is called after an array operation, the row count received is the aggregate number of rows affects by all the elements in the input parameter value array.

When connected to DB2 Universal Database, the application has the option of ATOMIC or NOT ATOMIC COMPOUND SQL. With ATOMIC SQL (the default) either all the elements of the array are processed successfully, or none at all. The application can choose to select the type of COMPOUND SQL used by setting the SQL_ATTR_PARAMOPT_ATOMIC attribute with SQLSetStmtAttr().

Note: SQLBindParam() must not be used to bind an array storage location to a parameter marker. In the case of character or binary input data, there is no method to specify the size of each element in the input array.

For queries with parameter markers on the WHERE clauses, an array of input values will cause multiple sequential result sets to be generated. Each result set can be processed before moving onto the next one by calling SQLMoreResults(). See “SQLMoreResults - Determine If There Are More Result Sets” on page 562 for more information and an example.

Parameter Binding Offsets

When an application needs to change parameter bindings it can call SQLBindParameter() a second time. This will change the bound parameter buffer address and the corresponding length/indicator buffer address used. This can only be used with row wise array inserts, but will work whether the application binds parameters individually or using an array.

Instead of multiple calls to SQLBindParameter(), DB2 CLI also supports parameter binding offsets. Rather than re-binding each time, an offset can be used to specify new buffer and length/indicator addresses which will be used in a subsequent call to SQLExecute() or SQLExecDirect().

To make use of parameter binding offsets, an application would follow these steps:

1. Call SQLBindParameter() as usual. The first set of bound parameter buffer addresses and the corresponding length/indicator buffer addresses will act as a template. The application will then move this template to different memory locations using the offset.
2. Call SQLExecute() or SQLExecDirect() as usual. The values stored in the bound addresses will be used.
3. Set up a variable to hold the memory offset value.

   The statement attribute SQL_ATTR_PARAM_BIND_OFFSET_PTR points to the address of an SQLINTEGER buffer where the offset will be stored. This address must remain valid until the cursor is closed.

   This extra level of indirection enables the use of a single memory variable to store the offset for multiple sets of parameter buffers on different statement handles. The application need only set this one memory variable and all of the offsets will be changed.
4. Store an offset value (number of bytes) in the memory location pointed to by the statement attribute set in the previous step.

   The offset value is always added to the memory location of the originally bound values. This sum must point to a valid memory address.
5. Call SQLExecute() or SQLExecDirect() again. CLI will add the offset specified above to the locations used in the original call to SQLBindParam() to determine where in memory to find the parameters to use.
6. Repeat steps 4 and 5 above as required.

See the section “Parameter Binding Offsets” on page 259 in SQLBindParam() for more information.

Array Input Example

This example shows an array INSERT statement, for an example of an array query statement, refer to “SQLMoreResults - Determine If There Are More Result Sets” on page 562.

```
/* From CLI sample custin.c */
/* ... */
    SQLCHAR * stmt =
    "INSERT INTO CUSTOMER ( Cust_Num, First_Name, Last_Name ) "
    "VALUES (?, ?, ?)" ;
```

```
    SQLINTEGER Cust_Num[] = {
        10,  20,  30,  40,  50,  60,  70,  80,  90, 100,
       110, 120, 130, 140, 150, 160, 170, 180, 190, 200,
       210, 220, 230, 240, 250,
    } ;

    SQLCHAR First_Name[][31] = {
       "EVA",     "EILEEN",    "THEODORE", "VINCENZO",  "SEAN",
       "DOLORES", "HEATHER",   "BRUCE",    "ELIZABETH", "MASATOSHI",
       "MARILYN", "JAMES",     "DAVID",    "WILLIAM",   "JENNIFER",
       "JAMES",   "SALVATORE", "DANIEL",   "SYBIL",     "MARIA",
       "ETHEL",   "JOHN",      "PHILIP",   "MAUDE",     "BILL",
    } ;

    SQLCHAR Last_Name[][31] = {
       "SPENSER", "LUCCHESI", "O'CONNELL", "QUINTANA",  "NICHOLLS",
       "ADAMSON", "PIANKA",   "YOSHIMURA", "SCOUTTEN",  "WALKER",
       "BROWN",   "JONES",    "LUTZ",      "JEFFERSON", "MARINO",
       "SMITH",   "JOHNSON",  "PEREZ",     "SCHNEIDER", "PARKER",
       "SMITH",   "SETRIGHT", "MEHTA",     "LEE",       "GOUNOT",
    } ;

/* ... */
    /* Prepare the statement */
    rc = SQLPrepare( hstmt, stmt, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLSetStmtAttr( hstmt,
                         SQL_ATTR_PARAMSET_SIZE,
                         ( SQLPOINTER ) row_array_size,
                         0
                       ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindParameter( hstmt,
                           1,
                           SQL_PARAM_INPUT,
                           SQL_C_SLONG,
                           SQL_INTEGER,
                           0,
                           0,
                           Cust_Num,
                           0,
                           NULL
                          ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindParameter( hstmt,
                           2,
                           SQL_PARAM_INPUT,
                           SQL_C_CHAR,
                           SQL_CHAR,
                           31,
                           0,
                           First_Name,
```

```
                        31,
                        NULL
                      ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindParameter( hstmt,
                        3,
                        SQL_PARAM_INPUT,
                        SQL_C_CHAR,
                        SQL_CHAR,
                        31,
                        0,
                        Last_Name,
                        31,
                        NULL
                      ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLExecute( hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
printf( "Inserted %ld Rows\n", row_array_size ) ;
```

Retrieving a Result Set into an Array

One of the most common tasks performed by an application is to issue a query statement, and then fetch each row of the result set into application variables that have been bound using SQLBindCol(). If the application requires that each column or each row of the result set be stored in an array, each fetch must be followed by either a data copy operation or a new set of SQLBindCol() calls to assign new storage areas for the next fetch.

Alternatively, applications can eliminate the overhead of extra data copies or extra SQLBindCol() calls by retrieving multiple rows of data (called a rowset) at a time into an array.

Note: A third method of reducing overhead, which can be used on its own or with arrays, is to specify a binding offset. Rather than re-binding each time, an offset can be used to specify new buffer and length/indicator addresses which will be used in a subsequent call to SQLFetch() or SQLFetchScroll(). This can only be used with row offset binding, and is described in “Column Binding Offsets” on page 93.

When retrieving a result set into an array, SQLBindCol() is also used to assign storage for application array variables. By default, the binding of rows is in column-wise fashion: this is symmetrical to using SQLBindParameter() to bind arrays of input parameter values as described in the previous section.

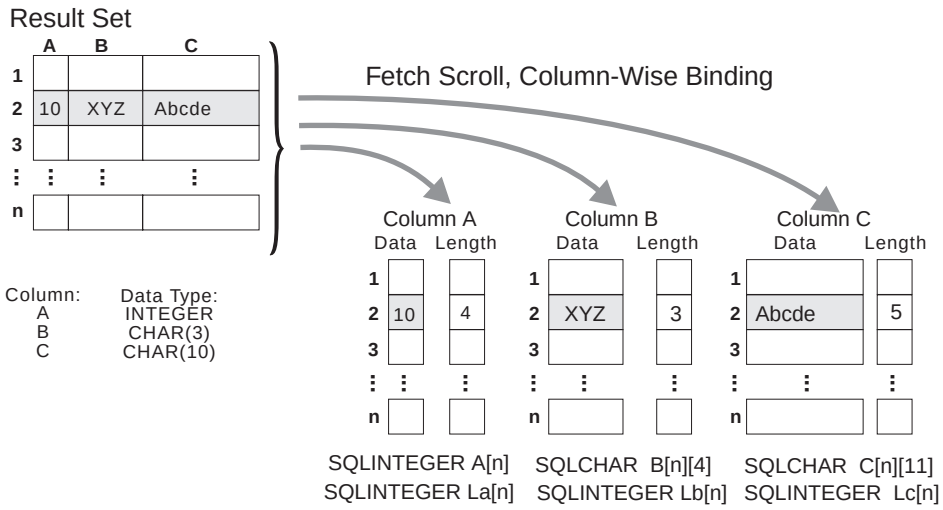


Figure 12. Column-Wise Binding

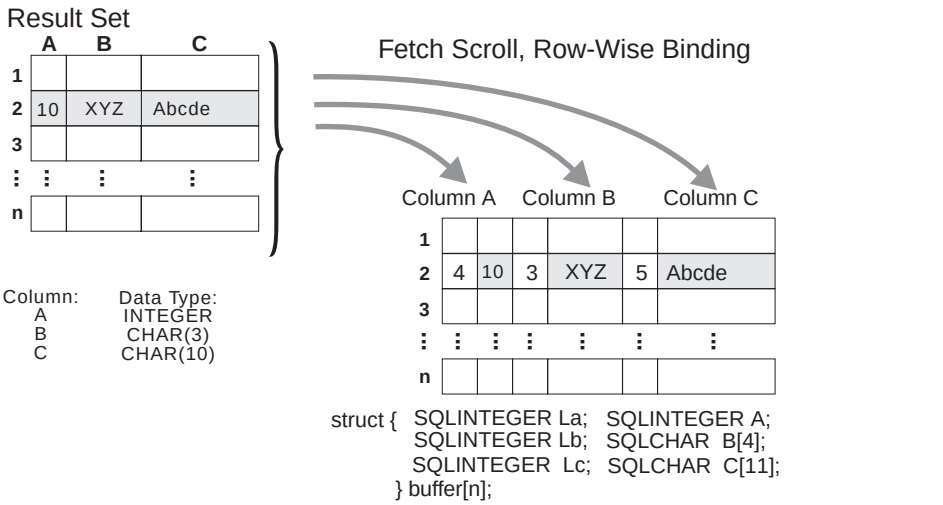


Figure 13. Row-Wise Binding

`SQLFetchScroll()` supports scrollable cursors, the ability to move forwards and backwards from any position in the result set. This can be used with both column wise and row wise binding. See “Scrollable Cursors” on page 68 for more information.

Returning Array Data for Column-Wise Bound Data

Figure 12 on page 91 is a logical view of column-wise binding. The right side of Figure 14 on page 93 shows the function flows for column-wise retrieval.

To specify column-wise array retrieval, the application calls `SQLSetStmtAttr()` with the SQL_ATTR_ROW_ARRAY_SIZE attribute to indicate how many rows to retrieve at a time. When the value of the SQL_ATTR_ROW_ARRAY_SIZE attribute is greater than 1, DB2 CLI knows to treat the deferred output data pointer and length pointer as pointers to arrays of data and length rather than to one single element of data and length of a result set column.

The application then calls `SQLFetchScroll()` to retrieve the data. When returning data, DB2 CLI uses the maximum buffer size argument (*BufferLength*) on `SQLBindCol()` to determine where to store successive rows of data in the array; the number of bytes available for return for each element is stored in the deferred length array. If the number of rows in the result set is greater than the SQL_ATTR_ROW_ARRAY_SIZE attribute value, multiple calls to `SQLFetchScroll()` are required to retrieve all the rows.

Returning Array Data for Row-Wise Bound Data

The application can also do row-wise binding which associates an entire row of the result set with a structure. In this case the rowset is retrieved into an array of structures, each of which holds the data in one row and the associated length fields. Figure 13 on page 91 gives a pictorial view of row-wise binding.

To perform row-wise array retrieval, the application needs to call `SQLSetStmtAttr()` with the SQL_ATTR_ROW_ARRAY_SIZE attribute to indicate how many rows to retrieve at a time. In addition, it must call `SQLSetStmtAttr()` with the SQL_ATTR_ROW_BIND_TYPE attribute value set to the size of the structure to which the result columns will be bound. DB2 CLI treats the deferred output data pointer of `SQLBindCol()` as the address of the data field for the column in the first element of the array of these structures. It treats the deferred output length pointer as the address of the associated length field of the column.

The application then calls `SQLFetchScroll()` to retrieve the data. When returning data, DB2 CLI uses the structure size provided with the SQL_ATTR_ROW_BIND_TYPE attribute to determine where to store successive rows in the array of structures.

Figure 14 shows the required functions for each method. The left side shows *n* rows being selected, and retrieved one row at a time into m application variables. The right side shows the same n rows being selected, and retrieved directly into an array.

- The diagram shows *m* columns bound, so *m* calls to `SQLBindCol()` are required in both cases.
- If arrays of less than *n* elements had been allocated, then multiple `SQLFetchScroll()` calls would be required.

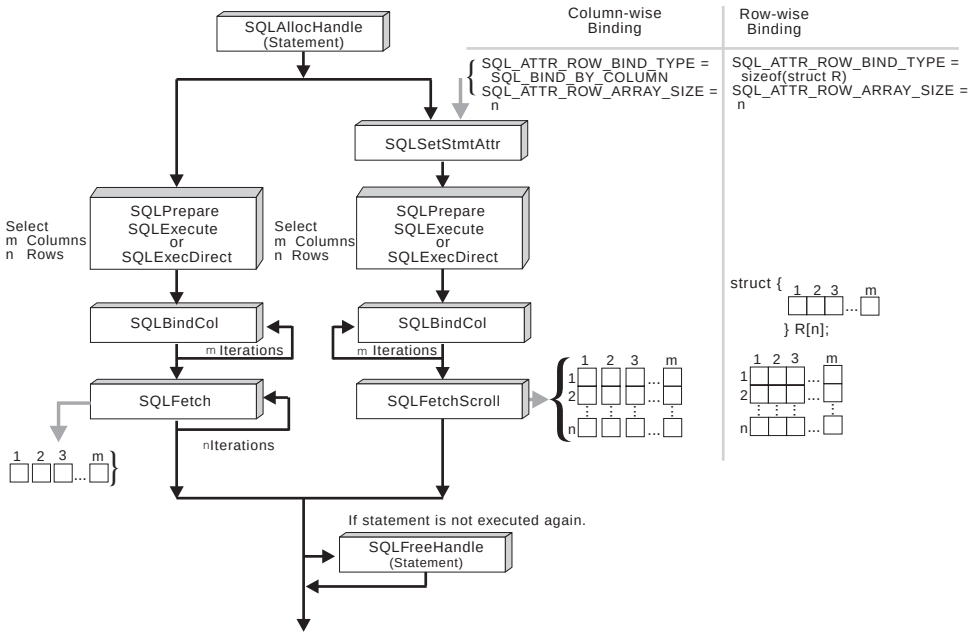


Figure 14. Array Retrieval

Column Binding Offsets

When an application needs to change bindings (for a subsequent fetch for example) it can call `SQLBindCol()` a second time. This will change the buffer address and length/indicator pointer used.

Instead of multiple calls to `SQLBindCol()`, DB2 CLI also supports column binding offsets. Rather than re-binding each time, an offset can be used to specify new buffer and length/indicator addresses which will be used in a subsequent call to `SQLFetch()` or `SQLFetchScroll()`. This can only be used with row wise binding, but will work whether the application retrieves a single row or multiple rows at a time.

To make use of column binding offsets, an application would follow these steps:

1. Call SQLBindCol() as usual. The first set of bound data buffer and length/indicator buffer addresses will act as a template. The application will then move this template to different memory locations using the offset.
2. Call SQLFetch() or SQLFetchScroll() as usual. The data returned will be stored in the locations bound above.
3. Set up a variable to hold the memory offset value.

   The statement attribute SQL_ATTR_ROW_BIND_OFFSET_PTR points to the address of an SQLINTEGER buffer where the offset will be stored. This address must remain valid until the cursor is closed.

   This extra level of indirection enables the use of a single memory variable to store the offset for multiple sets of bindings on different statement handles. The application need only set this one memory variable and all of the offsets will be changed.
4. Store an offset value (number of bytes) in the memory location pointed to by the statement attribute set in the previous step.

   The offset value is always added to the memory location of the originally bound values. This sum must point to a valid memory address.
5. Call SQLFetch() or SQLFetchScroll() again. CLI will add the offset specified above to the locations used in the original call to SQLBindCol() to determine where in memory to store the results.
6. Repeat steps 4 and 5 above as required.

See the section "Column Binding Offsets" on page 232 in SQLBindCol() for more information.

Column-Wise, Row-Wise Binding Example

```
/* From CLI sample ordrep.c */
/* ... */

    SQLCHAR * stmt =
       /* Common Table expression (or Define Inline View) */
       "WITH order (ord_num, cust_num, prod_num, quantity, amount) AS ( "
          "SELECT c.ord_num, c.cust_num, l.prod_num, l.quantity, "
                 "price(char(p.price, '.'), p.units, char(l.quantity, '.')) "
          "FROM ord_cust c, ord_line l, product p "
          "WHERE c.ord_num = l.ord_num "
                "AND l.prod_num = p.prod_num "
                "AND cast (cust_num as integer) = ? "
       "), "
       "totals (ord_num, total) AS ( "
          "SELECT ord_num, sum(decimal(amount, 10, 2)) "
          "FROM order GROUP BY ord_num "
       ") "
```

```
      /* The 'actual' SELECT from the inline view */
      "SELECT order.ord_num, cust_num, prod_num, quantity, "
             "DECIMAL(amount,10,2) amount, total "
      "FROM order, totals "
      "WHERE order.ord_num = totals.ord_num" ;

   /* Array of customers to get list of all orders for */
   SQLINTEGER Cust[] = {
       10,  20,  30,  40,  50,  60,  70,  80,  90, 100,
      110, 120, 130, 140, 150, 160, 170, 180, 190, 200,
      210, 220, 230, 240, 250,
   } ;

   /* Row-Wise (Includes buffer for both column data and length) */
   typedef struct {
      SQLINTEGER Ord_Num_L ;
      SQLINTEGER Ord_Num ;
      SQLINTEGER Cust_Num_L ;
      SQLINTEGER Cust_Num ;
      SQLINTEGER Prod_Num_L ;
      SQLINTEGER Prod_Num ;
      SQLINTEGER Quant_L ;
      SQLDOUBLE  Quant ;
      SQLINTEGER Amount_L ;
      SQLDOUBLE  Amount ;
      SQLINTEGER Total_L ;
      SQLDOUBLE  Total ;
   }  ord_info ;
   ord_info ord_array[row_array_size] ;

   SQLUINTEGER num_rows_fetched ;
   SQLUSMALLINT row_status_array[row_array_size], i, j ;

/* ... */
   /* Get details and total for each order Row-Wise */
   rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
   CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

   rc = SQLSetStmtAttr( hstmt,
                        SQL_ATTR_PARAMSET_SIZE,
                        ( SQLPOINTER ) row_array_size,
                        0
                      ) ;
   CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

   rc = SQLBindParameter( hstmt,
                          1,
                          SQL_PARAM_INPUT,
                          SQL_C_LONG,
                          SQL_INTEGER,
                          0,
                          0,
                          Cust,
                          0,
                          NULL
```

```
                        ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLExecDirect( hstmt, stmt, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* SQL_ROWSET_SIZE sets the max number of result rows to fetch each time */
    rc = SQLSetStmtAttr( hstmt,
                         SQL_ATTR_ROW_ARRAY_SIZE,
                         ( SQLPOINTER ) row_set_size,
                         0
                       ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* Set Size of One row, Used for Row-Wise Binding Only */
    rc = SQLSetStmtAttr( hstmt,
                         SQL_ATTR_ROW_BIND_TYPE,
                         ( SQLPOINTER ) sizeof( ord_info ) ,
                         0
                       ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLSetStmtAttr( hstmt,
                         SQL_ATTR_ROW_STATUS_PTR,
                         ( SQLPOINTER ) row_status_array,
                         0
                       ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLSetStmtAttr( hstmt,
                         SQL_ATTR_ROWS_FETCHED_PTR,
                         ( SQLPOINTER ) &num_rows_fetched,
                         0
                       ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* Bind column 1 to the Ord_num Field of the first row in the array */
    rc = SQLBindCol( hstmt,
                     1,
                     SQL_C_LONG,
                     ( SQLPOINTER ) & ord_array[0].Ord_Num,
                     0,
                     &ord_array[0].Ord_Num_L
                   ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* Bind remaining columns ... */
/* ... */
    /*
     NOTE: This sample assumes that an order will never have more
           rows than row_set_size.  A check should be added below to call
           SQLExtendedFetch multiple times for each result set.
    */

    while ( SQLFetchScroll( hstmt, SQL_FETCH_NEXT, 0 ) != SQL_NO_DATA ) {
```

```
    printf( "************************************\n" ) ;
    printf( "Orders for Customer: %ld\n", ord_array[0].Cust_Num ) ;
    printf( "************************************\n" ) ;
    i = 0 ;
    while ( i < num_rows_fetched ) {
       if ( row_status_array[i] == SQL_ROW_SUCCESS ||
            row_status_array[i] == SQL_ROW_SUCCESS_WITH_INFO
          ) {
          printf( "\nOrder #: %ld\n", ord_array[i].Ord_Num ) ;
          printf( "     Product  Quantity         Price\n" ) ;
          printf( "     -------- ---------------- ------------\n" ) ;
          j = i ;
          while ( ord_array[j].Ord_Num == ord_array[i].Ord_Num ) {
             printf( "    %8ld %16.7lf %12.2lf\n",
                     ord_array[i].Prod_Num,
                     ord_array[i].Quant,
                     ord_array[i].Amount
                   ) ;
             i++ ;
             if ( i >= num_rows_fetched ) break ;
             if ( row_status_array[i] != SQL_ROW_SUCCESS )
                if ( row_status_array[i] != SQL_ROW_SUCCESS_WITH_INFO )
                   break ;
          }
          printf( "                                  ============\n" ) ;
          printf( "                                 %12.2lf\n",
                  ord_array[j].Total
                ) ;
       }
       else i++ ;
    }
}
```

Using Descriptors

DB2 CLI stores information (data types, size, pointers, and so on) about columns in a result set, and parameters in an SQL statement. The bindings of application buffers to columns and parameters must also be stored. *Descriptors* are a logical view of this information, and provide a way for applications to query and update this information.

Many CLI functions make use of descriptors, but the application itself does not need to manipulate them directly.

For instance:

- When an application binds column data using `SQLBindCol()` descriptor fields are set that completely describe the binding.
- A number of statement attributes correspond to the header fields of a descriptor. In this case you can achieve the same effect calling

SQLSetStmtAttr() as calling the corresponding function SQLSetDescField() that sets the values in the descriptor directly.

Although no database operations require direct access to descriptors, there are situations where working directly with the descriptors will be more efficient or result in simpler code. For instance, a descriptor that describes a row fetched from a table can then be used to describe a row inserted back into the table.

Descriptor Types

There are four types of descriptors, as follows:

Application Parameter Descriptor (APD)
Describes the application buffers (pointers, data types, scale, precision, length, maximum buffer length, and so on) that are bound to parameters in an SQL statement. If the parameters are part of a CALL statement they may be input, ouput, or both. This information is described using the application's C data types.

Application Row Descriptor (ARD)
Describes the application buffers bound to the columns. The application may specify different data types from those in the implementation row descriptor to achieve data conversion of column data. This descriptor reflects any data conversion that the application may specify.

Implementation Parameter Descriptor (IPD)
Describes the parameters in the SQL statement (SQL type, size, precision, and so on).

- If the parameter is used as input, this describes the SQL data that the database server will recive after DB2 CLI has performed any required conversion.
- If the parameter is used as output, this describes the SQL data before DB2 CLI performs any required conversion to the application's C data types.

Implementation Row Descriptor (IRD)
Describes the row of data from the result set before DB2 CLI performs any required data conversion to the application's C data types.

The only difference between the four types of descriptors described above is how they are used. One of the benefits of descriptors is that a single descriptor can be used to serve multiple purposes. For instance, a row descriptor in one statement can be used as a parameter descriptor in another statement.

As soon as a descriptor exists it is either an application descriptor or an implementation descriptor. This is the case even if the descriptor has not yet been used in a database operation. If the descriptor is allocated by the application using SQLAllocHandle() then it is an application descriptor.

Values Stored in a Descriptor

Each descriptor contains both header fields and record fields. These fields together completely describe the column or parameter.

Header Fields

Each header field occurs once in each descriptor. Changing one of these fields affects all columns or parameters.

Many of the following header fields correspond to a statement attribute. Setting the header field of the descriptor using SQLSetDescField() is the same as setting the corresponding statement attribute using SQLSetStmtAttr(). The same holds true for retrieving the information using SQLGetDescField() or SQLGetStmtAttr(). If your application does not already have a descriptor handle allocated then it is more efficient to use the statement attribute calls instead of allocating the descriptor handle then using the descriptor calls.

Table 7. Header fields

SQL_DESC_ALLOC_TYPE	SQL_DESC_BIND_TYPE[a]
SQL_DESC_ARRAY_SIZE[a]	SQL_DESC_COUNT
SQL_DESC_ARRAY_STATUS_PTR[a]	SQL_DESC_ROWS_PROCESSED_PTR[a]
SQL_DESC_BIND_OFFSET_PTR[a]	

Note:

[a] Header field that corresponds to a statement attribute.

For more information about each of these fields see “Header Fields” on page 659 in SQLSetDescField().

The descriptor header field SQL_DESC_COUNT is the one-based index of the highest-numbered descriptor record that contains information. DB2 CLI automatically updates this field (and the physical size of the desciptor) as columns or parameters are bound and unbound. The initial value of SQL_DESC_COUNT is 0 when a descriptor is first allocated.

Descriptor Records

Zero or more descriptor records are contained in a single descriptor. As new columns or parameters are bound, new desciptor records are added to the descriptor. When a column or parameter is unbound, the descriptor record is removed.

Table 8 lists the fields in a descriptor record. They describe a column or parameter, and occur once in each descriptor record.

Table 8. Record Fields

SQL_DESC_AUTO_UNIQUE_VALUE	SQL_DESC_LOCAL_TYPE_NAME
SQL_DESC_BASE_COLUMN_NAME	SQL_DESC_NAME
SQL_DESC_BASE_TABLE_NAME	SQL_DESC_NULLABLE
SQL_DESC_CASE_SENSITIVE	SQL_DESC_OCTET_LENGTH
SQL_DESC_CATALOG_NAME	SQL_DESC_OCTET_LENGTH_PTR
SQL_DESC_CONCISE_TYPE	SQL_DESC_PARAMETER_TYPE
SQL_DESC_DATA_PTR	SQL_DESC_PRECISION
SQL_DESC_DATETIME_INTERVAL_CODE	SQL_DESC_SCALE
SQL_DESC_DATETIME_INTERVAL_PRECISION	SQL_DESC_SCHEMA_NAME
SQL_DESC_DISPLAY_SIZE	SQL_DESC_SEARCHABLE
SQL_DESC_FIXED_PREC_SCALE	SQL_DESC_TABLE_NAME
SQL_DESC_INDICATOR_PTR	SQL_DESC_TYPE
SQL_DESC_LABEL	SQL_DESC_TYPE_NAME
SQL_DESC_LENGTH	SQL_DESC_UNNAMED
SQL_DESC_LITERAL_PREFIX	SQL_DESC_UNSIGNED
SQL_DESC_LITERAL_SUFFIX	SQL_DESC_UPDATABLE

For more information about each of these fields see “Record Fields” on page 665 in `SQLSetDescField()`.

Deferred Fields: Deferred fields are created when the descriptor header or a descriptor record is created. The addresses of the defined variables are stored but not used until a later point in the application. The application must not deallocate or discard these variables between the time it associates them with the fields and the time CLI reads or writes them.

The following table lists the deferred fields and the meaning or a null pointer where applicable:

Table 9. Deferred Fields

Field	Meaning of Null value
SQL_DESC_DATA_PTR	The record is unbound.
SQL_DESC_INDICATOR_PTR	(none)

Table 9. Deferred Fields (continued)

Field	Meaning of Null value
SQL_DESC_OCTET_LENGTH_PTR (ARD and APD only)	• ARD: The length information for that column is not returned. • APD: If the parameter is a character string, the driver assumes that string is null-terminated. For output parameters, a null value in this field prevents the driver from returning length information. (If the SQL_DESC_TYPE field does not indicate a character-string parameter, the SQL_DESC_OCTET_LENGTH_PTR field is ignored.)
SQL_DESC_ARRAY_STATUS_PTR (multirow fetch only)	A multirow fetch failed to return this component of the per-row diagnostic information.
SQL_DESC_ROWS_PROCESSED_PTR (multirow fetch only)	(none)

Bound Descriptor Records: The SQL_DESC_DATA_PTR field in each descriptor record points to a variable that contains the parameter value (for APDs) or the column value (for ARDs). This is a deferred field that defaults to null. Once the column or parameter is bound it points to the parameter or column value. At this point the descriptor record is said to be bound.

Application Parameter Descriptors (APD)
Each bound record constitutes a bound parameter. The application must bind a parameter for each input and output parameter marker in the SQL statement before the statement is executed.

Application Row Descriptors (ARD)
Each bound record relates to a bound column.

Consistency Check: A consistency check is performed automatically whenever an application sets the SQL_DESC_DATA_PTR field of the APD or ARD. The check ensures that various fields are consistent with each other, and that appropriate data types have been specified.

To force a consistency check of IPD fields, the application can set the SQL_DESC_DATA_PTR field of the IPD. This setting is only used to force the consistency check. The value is not stored and cannot be retrieved by a call to `SQLGetDescField()` or `SQLGetDescRec()`.

A consistency check cannot be performed on an IRD.

See “Consistency Checks” on page 679 in SQLSetDescRec() for more information on the consistency check.

Allocating and Freeing Descriptors

Descriptors are allocated in one of two ways:

Implicitly Allocated Descriptors

When a statement handle is allocated, a set of four descriptors are implicitly allocated. When the statement handle is freed, all implicitly allocated descriptors on that handle are freed as well.

To obtain handles to these implicitly allocated descriptors an application can call SQLGetStmtAttr(), passing the statement handle and an *Attribute* value of:

- SQL_ATTR_APP_PARAM_DESC (APD)
- SQL_ATTR_APP_ROW_DESC (ARD)
- SQL_ATTR_IMP_PARAM_DESC (IPD)
- SQL_ATTR_IMP_ROW_DESC (IRD)

Explicitly Allocated Descriptors

An application can explicitly allocate application descriptors. It is not possible, however, to allocate implementation descriptors.

An application descriptor on a connection can be explicitly allocated at any time it is connected to the database. This is done by calling SQLSetStmtAttr(), passing the statement handle and an *Attribute* value of:

- SQL_ATTR_APP_PARAM_DESC (APD)
- SQL_ATTR_APP_ROW_DESC (ARD)

In this case the explicitly specified allocated descriptor will be used rather than the implicitly allocated descriptor.

An explicitly allocated descriptor can be associated with more than one statement.

Initialization of Fields

When an application row descriptor is allocated, its fields receive the initial values indicated in the “Initialization of Descriptor Fields” on page 651 section of SQLSetDescField(). The SQL_DESC_TYPE field is set to SQL_DEFAULT which provides for a standard treatment of database data for presentation to the application. The application may specify different treatment of the data by setting fields of the descriptor record.

The initial value of the SQL_DESC_ARRAY_SIZE header field is 1. To enable multirow fetch, the application can set this value in an ARD to specify the number of rows in a rowset. See “Scrollable Cursors” on page 68 for information about rowsets in a scrollable cursor.

There are no default values for the fields of an IRD. The fields are set when there is a prepared or executed statement.

The following fields in an IPD are undefined until they have been automatically populated by a call to SQLPrepare():

- SQL_DESC_CASE_SENSITIVE
- SQL_DESC_FIXED_PREC_SCALE
- SQL_DESC_TYPE_NAME
- SQL_DESC_DESC_UNSIGNED
- SQL_DESC_LOCAL_TYPE_NAME

Automatic Population of the IPD

There are times when the application will need to discover information about the parameters of a prepared SQL statement. A good example is when an ad-hoc query is prepared; the application will not know anything about the parameters in advance. If the application enables automatic population of the IPD, by setting the SQL_ATTR_ENABLE_AUTO_IPD statement attribute to SQL_TRUE (using SQLSetStmtAttr()), then the fields of the IPD are automatically populated to describe the parameter. This includes the data type, precision, scale, and so on (the same information that SQLDescribeParam() returns). The application can use this information to determine if data conversion is required, and which application buffer is the most appropriate to bind the parameter to.

Automatic population of the IPD involves some overhead. If it is not necessary for this information to be automatically gathered by the CLI driver then the SQL_ATTR_ENABLE_AUTO_IPD statement attribute should be set to SQL_FALSE. This is the default setting, and the application should return it to this value when it is no longer needed.

When automatic population of the IPD is active, each call to SQLPrepare() causes the fields of the IPD to be updated. The resulting descriptor information can be retrieved by calling the following functions:

- SQLGetDescField()
- SQLGetDescRec()
- SQLDescribeParam()

Freeing Descriptors

Explicitly Allocated Descriptors

When an explicitly allocated descriptor is freed, all statement handles to which the freed descriptor applied automatically revert to the original descriptors implicitly allocated for them.

Explicitly allocated descriptors can be freed in one of two ways:

- by calling `SQLFreeHandle()` with a *HandleType* of SQL_HANDLE_DESC
- by freeing the connection handle that the descriptor is associated with

Implicitly Allocated Descriptors

An implicitly allocated descriptor can be freed in one of the following ways:

- by calling `SQLDisconnect()` which drops any statements or descriptors open on the connection
- by calling `SQLFreeHandle()` with a *HandleType* of SQL_HANDLE_STMT to free the statement handle and all of the implicitly allocated descriptors associated with the statement

An implicitly allocated descriptor cannot be freed by calling `SQLFreeHandle()` with a *HandleType* of SQL_HANDLE_DESC.

Getting, Setting, and Copying Descriptor Fields

The following sections describe manipulating descriptors using descriptor handles. The final section, "Accessing Descriptors without using a Handle" on page 106 describes how to manipulate descriptor values by calling CLI functions that do not use descriptor handles.

The handle of an explicitly allocated descriptor is returned in the *OutputHandlePtr* argument when the application calls `SQLAllocHandle()` to allocate the descriptor.

The handle of an implicitly allocated descriptor is obtained by calling `SQLGetStmtAttr()` with either SQL_ATTR_IMP_PARAM_DESC or SQL_ATTR_IMP_ROW_DESC.

Retrieving Values in Descriptor Fields

See "SQLGetDescField - Get Single Field Settings of Descriptor Record" on page 458 for information on how to obtain a single field of a descriptor record.

See “SQLGetDescRec - Get Multiple Field Settings of Descriptor Record” on page 463 for information on how to obtain the settings of multiple descriptor fields that affect the data type and storage of column or parameter data.

Setting Values of Descriptor Fields

This section deals with how to set the values of descriptor fields using descriptor handles. You can also set many of these fields without using descriptor handles; see “Accessing Descriptors without using a Handle” on page 106 for more information.

Two methods can be used to set descriptor fields, one field at a time or multiple fields at a time:

Setting Descriptor Fields Individually: Some fields of a descriptor are read-only, but the others can be set using the function `SQLSetDescField()`. See the following sections for specific details on each field that can be set:

- “Header Fields” on page 659
- “Record Fields” on page 665

Record and header fields are set differently using `SQLSetDescField()`:

Header fields
The call to `SQLSetDescField()` passes the header field to be set and a record number of 0. The record number is ignored since there is only one header field per descriptor. In this case the record number of 0 does not indicate the bookmark field.

Record fields
The call to `SQLSetDescField()` passes the record field to be set and a record number of 1 or higher, or 0 to indicate the bookmark field.

The application must follow the steps defined in “Sequence of Setting Descriptor Fields” on page 651 when setting individual fields of a descriptor. Setting some fields will cause DB2 CLI to automatically set other fields. A consistency check will take place after the application follows the defined steps. This will ensure that the values in the descriptor fields are consistent. See “Consistency Check” on page 101 for more information.

If a function call that would set a descriptor fails, the contents of the descriptor fields are undefined after the failed function call.

Setting Multiple Descriptor Fields at a time: A predefined set of descriptor fields can be set with one call rather than setting individual fields one at a time. `SQLSetDescRec()` sets the following fields for a single column or parameter:

- SQL_DESC_TYPE
- SQL_DESC_OCTET_LENGTH
- SQL_DESC_PRECISION
- SQL_DESC_SCALE
- SQL_DESC_DATA_PTR
- SQL_DESC_OCTET_LENGTH_PTR
- SQL_DESC_INDICATOR_PTR

(SQL_DESC_DATETIME_INTERVAL_CODE is also defined by ODBC but is not supported by DB2 CLI.)

See "SQLSetDescRec - Set Multiple Descriptor Fields for a Column or Parameter Data" on page 677 for more information.

Copying Descriptors

One benefit of descriptors is the fact that a single descriptor can be used for multiple purposes. For instance, an ARD on one statement handle can be used as an APD on another statement handle.

There will be other instances, however, where the application will want to make a copy of the original descriptor, then modify certain fields. In this case `SQLCopyDesc()` is used to overwrite the fields of an existing descriptor with the values from another descriptor. Only fields that are defined for both the source and target descriptors are copied (with the exception of the SQL_DESC_ALLOC_TYPE field which cannot be changed).

Fields can be copied from any type of descriptor, but can only be copied to an application descriptor (APD or ARD) or an IPD. Fields cannot be copied to an IRD. The descriptor's allocation type will not be changed by the copy procedure (again, the SQL_DESC_ALLOC_TYPE field cannot be changed).

See "SQLCopyDesc - Copy Descriptor Information Between Handles" on page 328 for complete details on copying descriptors.

Accessing Descriptors without using a Handle

As was mentioned at the beginning of this section on descriptors, many CLI functions make use of descriptors, but the application itself does not need to manipulate them directly. Instead, the application can use a different function which will set or retrieve one or more fields of a descriptor as well as perform other functions. This category of CLI functions are called *concise* functions. `SQLBindCol()` is an example of a concise function that manipulates descriptor fields.

In addition to manipulating multiple fields, concise functions are called without explicitly specifying the descriptor handle. The application does not even need to retrieve the descriptor handle to use a concise function.

The following types of concise functions exist::

- The functions SQLBindCol() and SQLBindParameter() bind a column or parameter by setting the descriptor fields that correspond to their arguments. These functions also perform other tasks unrelated to descriptors.

  If required, an application can also use the descriptor calls directly to modify individual details of a binding. In this case the descriptor handle must be retrieved, and the functions SQLSetDescField() or SQLSetDescRec() called to modify the binding.
- The following functions always retrieve values in descriptor fields:
  - SQLColAttribute()
  - SQLDescribeCol()
  - SQLDescribeParam()
  - SQLNumParams()
  - SQLNumResultCols()
- The functions SQLSetDescRec() and SQLGetDescRec() set or get the multiple descriptor fields that affect the data type and storage of column or parameter data. A single call to SQLSetDescRec() can be used to change the values used in the binding of a column or parameter.
- The functions SQLSetStmtAttr() and SQLGetStmtAttr() modify or return descriptor fields in some cases, depending on which statement attribute is specified. See “Values Stored in a Descriptor” on page 99 for details.

Descriptor Sample

```
/* From CLI sample descrptr.c */
/* ... */
    SQLCHAR * sqlstmt =
       "SELECT deptname, location from org where division = ? " ;
/* ... */

    /* macro to initalize server, uid and pwd */
    INIT_UID_PWD ;

    /* allocate an environment handle */
    rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    /* allocate a connect handle, and connect */
    rc = DBconnect( henv, &hdbc ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
```

```
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

/* Use SQLGetStmtAttr() to get implicit parameter descriptor handle */
rc = SQLGetStmtAttr ( hstmt,
                      SQL_ATTR_IMP_PARAM_DESC,
                      &hIPDdesc,
                      SQL_IS_POINTER,
                      NULL);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Use SQLGetStmtAttr() to get implicit row descriptor handle */
rc = SQLGetStmtAttr ( hstmt,
                      SQL_ATTR_IMP_ROW_DESC,
                      &hIRDdesc,
                      SQL_IS_POINTER,
                      NULL);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Call SQLGetDescField() to see how the header field */
/* SQL_DESC_ALLOC_TYPE is set. */
rc = SQLGetDescField( hIPDdesc,
                      0, /* ignored for header fields */
                      SQL_DESC_ALLOC_TYPE,
                      &desc_smallint, /* The result */
                      SQL_IS_SMALLINT,
                      NULL ); /* ignored */
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Print the descriptor information */
printf("The IPD header descriptor field SQL_DESC_ALLOC_TYPE is %s\n",
        ALLOCTYPES[desc_smallint]);

/* prepare statement for multiple use */
rc = SQLPrepare(hstmt, sqlstmt, SQL_NTS);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* bind division to parameter marker in sqlstmt */
rc = SQLBindParameter( hstmt,
                       1,
                       SQL_PARAM_INPUT,
                       SQL_C_CHAR,
                       SQL_CHAR,
                       10,
                       0,
                       division.s,
                       11,
                       NULL
                     ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* bind deptname to first column in the result set */
rc = SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) deptname.s, 15,
                &deptname.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

```
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) location.s, 14,
                &location.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Call SQLGetDescField() to see how the descriptor record */
/* field SQL_DESC_PARAMETER_TYPE is set */
rc = SQLGetDescField( hIPDdesc,
                      1, /* Look at the parameter */
                      SQL_DESC_PARAMETER_TYPE,
                      &desc_smallint, /* The result */
                      SQL_IS_SMALLINT,
                      NULL ); /* ignored */
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

printf("The IPD record descriptor field SQL_DESC_PARAMETER_TYPE is %s\n",
        PARAMTYPE[desc_smallint]);

strcpy( division.s,  "Eastern");
rc = SQLExecute(hstmt);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

printf("\nDepartments in %s Division:\n", division.s);
printf("Department       Location\n");
printf("-------------- -------------\n");

while ( ( rc = SQLFetch( hstmt ) ) == SQL_SUCCESS )
   CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
   printf( "%-14.14s %-13.13s \n", deptname.s, location.s ) ;
if ( rc != SQL_NO_DATA_FOUND )
   CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Print out some implementation row descriptor fields */
/* from the last SQLFetch() above */

for (colCount = 1; colCount <=2; colCount++) {
   printf("\nInformation for column %i\n",colCount);

   /* Call SQLGetDescField() to see how the descriptor record */
   /* field SQL_DESC_TYPE_NAME is set */
   rc = SQLGetDescField( hIRDdesc,
               colCount,
               SQL_DESC_TYPE_NAME, /* record field */
               desc_char, /* The result */
               25,
               NULL ); /* ignored */
   CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

   printf("  - IRD record descriptor field SQL_DESC_TYPE_NAME is %s\n",
        desc_char);

   /* Call SQLGetDescField() to see how the descriptor record */
   /* field SQL_DESC_LABEL is set */
   rc = SQLGetDescField( hIRDdesc,
               colCount,
               SQL_DESC_LABEL, /* record field */
```

```
                desc_char, /* The result */
                25,
                NULL ); /* ignored */
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

        printf("  - IRD record descriptor field SQL_DESC_LABEL is %s\n",
             desc_char);

    } /* End of the for statement */
```

Using Compound SQL

Compound SQL allows multiple statements to be grouped into a executable single block. This block of statements, together with any input parameter values, can then be executed in a single continuous stream, reducing the execution time and network traffic. Compound SQL is most often used to efficiently execute a series of INSERT, UPDATE and DELETE statements.

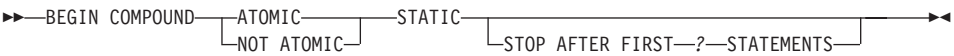
Any SQL statement that can be prepared dynamically, other than a query, can be executed as a statement inside a compound statement. Statements within a Compound SQL statement are referred to as sub-statements. Compound SQL does not guarantee the order in which the sub-statements are executed, therefore there must be no dependency between the statements.

Compound SQL statements cannot be nested. The authorization ID of the Compound SQL statement must be the appropriate authorization on all the individual sub-statements contained within the Compound SQL statement.

Compound SQL is supported when connected to DB2 Universal Database, or in DRDA environments with DB2 Connect V 2.3 or higher.

ATOMIC and NOT ATOMIC Compound SQL

A Compound SQL statement block is specified by surrounding the sub-statements by a BEGIN COMPOUND statement and an END COMPOUND statement. The BEGIN COMPOUND syntax is shown below:



ATOMIC
Specifies that, if any of the sub-statements within the Compound SQL statement fails, then all changes made to the database by any of the sub-statements are undone. ATOMIC is not supported in DRDA environments.

NOT ATOMIC
Specifies that, regardless of the failure of any sub-statements, the Compound SQL statement will not undo any changes made to the database by the other sub-statements.

STATIC
Specifies that input variables for all sub-statements retain their original value. If the same variable is set by more than one sub-statement, the value of that variable following the Compound SQL statement is the value set by the last sub-statement.

STOP AFTER FIRST ? STATEMENTS
Specifies that only a certain number of sub-statements are to be executed. If this clause is omitted, all the sub-statements are executed.

The END COMPOUND syntax is shown below:

Specifying the COMMIT option will commit all the sub-statements if they executed successfully. The COMMIT applies to the current transaction, including statements that precede the compound statement. If COMMIT is specified, and the connection is a coordinated distributed connection (SQL_COORDINATED_TRANS), an error will be returned (SQLSTATE of 25000).

If the COMMIT option is not specified after END COMPOUND, the sub-statements will not be committed unless the application is operating under auto-commit mode, in which case the commit will be issued at the END COMPOUND. For information on the auto-commit mode, refer to "Commit or Rollback" on page 22.

Figure 15 on page 113 shows the general sequence of function calls required to execute a compound SQL statement. Note that:

- `SQLPrepare()` and `SQLExecute()` can be used in place of `SQLExecDirect()`.
- The BEGIN COMPOUND and END COMPOUND statements are executed with the same statement handle.
- Each sub-statement must have its own statement handle.
- All statement handles must belong to the same connection, and have the same isolation level.
- The sub-statements should be, but do not need to be, prepared before the BEGIN COMPOUND statement, especially in DRDA environments where some optimization may be possible to reduce network flow.

- The statement handles must remain allocated until the END COMPOUND statement is executed.
- The only functions that may be called using the statement handles allocated for the compound sub-statements are:
  - SQLAllocHandle()
  - SQLBindParameter()
  - SQLBindFileToParam()
  - SQLParamData()
  - SQLPutData()
  - SQLExecDirect(), SQLPrepare(), SQLExecute()
- SQLTransact() cannot be called for the same connection, or any connect requests between BEGIN and END COMPOUND.
- The sub-statements may be executed in any order.
- SQLRowCount() (or SQLGetSQLCA()) can be called using the same statement handle as the BEGIN, END COMPOUND statement to get an aggregate count of the rows affected.

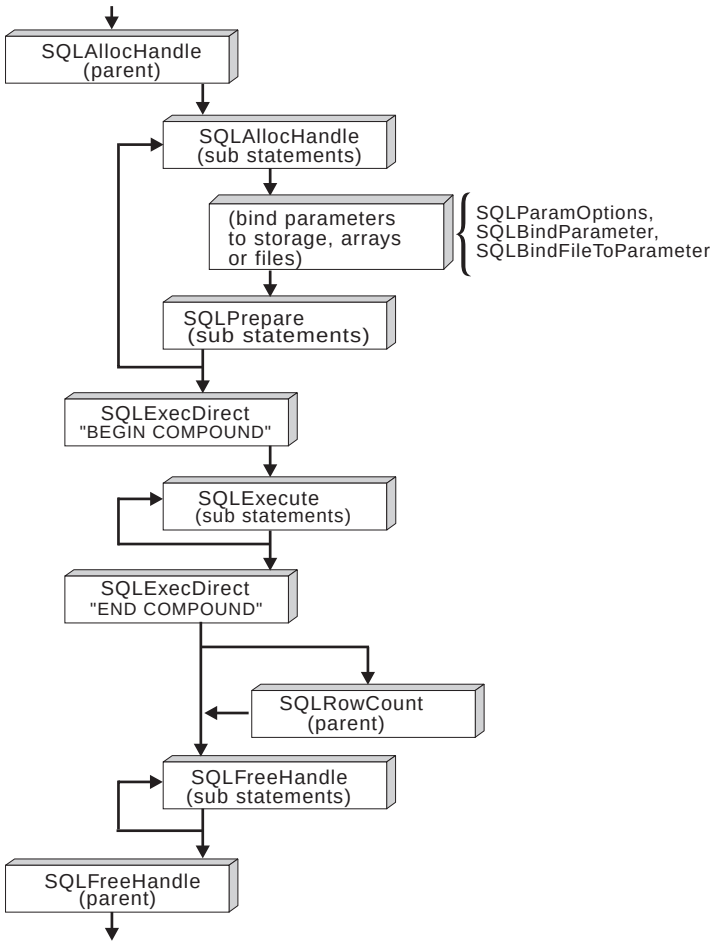


Figure 15. Compound SQL

Compound SQL Error Handling

If the compound statement is ATOMIC and the END COMPOUND `SQLExecDirect()` call returns:

- SQL_SUCCESS - all the sub-statements executed without any warnings or errors.
- SQL_SUCCESS_WITH_INFO - all the sub-statements executed successfully with one or more warnings. Call `SQLError()` to obtain generic diagnostic information, or call `SQLGetSQLCA()` to obtain the SQLCA for the entire

compound SQL statement. The statement handle used for SQLError() or SQLGetSQLCA() must the same one used to process the BEGIN, END COMPOUND SQL.

Most of the information in the SQLCA reflects values set by the database server when it processed the last sub-statement, such as the SQLCODE and SQLSTATE. If one or more error occurred and none of these are of a serious nature, the SQLERRMC field in the SQLCA will contain information on up to a maximum of seven of these errors.

- SQL_NO_DATA_FOUND - a BEGIN, END COMPOUND was executed without any sub-statements, or none of the sub-statement affected any rows.
- SQL_ERROR - one or more sub-statements failed, and all sub-statements were rolled back.

If the compound statement is NOT ATOMIC and the END COMPOUND SQLExecDirect() call returns:

- SQL_SUCCESS - all sub-statements executed without any errors.
- SQL_SUCCESS_WITH_INFO - the COMPOUND statement executed with one or more warnings. One or more sub-statements have returned an warning. Call SQLError() or SQLGetSQLCA() to receive additional information on the information on the warnings.
- SQL_NO_DATA_FOUND - a BEGIN, END COMPOUND was executed without any sub-statements, or none of the sub-statement affected any rows.
- SQL_ERROR - the COMPOUND statement failed. At least one sub-statement returned an error, examine the SQLCA to determine which statement(s) failed.

Note: Refer to the *SQL Reference* for details on the contents of an SQLCA after Compound SQL execution.

Compound SQL Example

The following example executes a compound statement consisting of 4 sub-statements to insert rows into a new AWARDS table.

```
/* From CLI sample compnd.c */
/* ... */
    SQLCHAR * stmt[] = {

        "INSERT INTO awards (id, award) "
        "SELECT id, 'Sales Merit' from staff "
        "WHERE job = 'Sales' AND (comm/100 > years)",

        "INSERT INTO awards (id, award) "
        "SELECT id, 'Clerk Merit' from staff "
        "WHERE job = 'Clerk' AND (comm/50 > years)",
```

```
        "INSERT INTO awards (id, award) "
        "SELECT id, 'Best ' concat job FROM STAFF "
        "WHERE comm = (SELECT max(comm) FROM staff WHERE job = 'Clerk')",

        "INSERT INTO awards (id, award) "
        "SELECT id, 'Best ' concat job FROM STAFF "
        "WHERE comm = (SELECT max(comm) FROM STAFF WHERE job = 'Sales')",

    } ;

    SQLINTEGER i ;

/* ... */
    /* Prepare 4 substatements */
    for ( i = 1; i < 4; i++ ) {
        rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &cmhstmt[i] ) ;
        CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;
        rc = SQLPrepare( cmhstmt[i], stmt[i], SQL_NTS ) ;
        CHECK_HANDLE( SQL_HANDLE_STMT, cmhstmt[i], rc ) ;
    }

    rc = SQLExecDirect( hstmt,
                        ( SQLCHAR * ) "BEGIN COMPOUND NOT ATOMIC STATIC",
                        SQL_NTS
                      ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* Execute 4 substatements */
    for ( i = 1; i < 4; i++ ) {
        rc = SQLExecute( cmhstmt[i] ) ;
        CHECK_HANDLE( SQL_HANDLE_STMT, cmhstmt[i], rc ) ;
    }

    /* Execute the COMPOUND statement (of 4 sub-statements) */
    printf( "Executing the COMPOUND statement (of 4 sub-statements)\n" ) ;

    rc = SQLExecDirect( hstmt,
                        ( SQLCHAR * ) "END COMPOUND COMMIT",
                        SQL_NTS
                      ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    for ( i = 1; i < 4; i++ ) {
        rc = SQLFreeHandle( SQL_HANDLE_STMT, cmhstmt[i] ) ;
        CHECK_HANDLE( SQL_HANDLE_STMT, cmhstmt[i], rc ) ;
    }
```

Using Large Objects

The term *large object* and the generic acronym *LOB* are used to refer to any type of large object. There are three LOB data types: Binary Large Object (BLOB), Character Large Object (CLOB), and Double-Byte Character Large Object (DBCLOB). These LOB data types are represented symbolically as SQL_BLOB, SQL_CLOB, SQL_DBCLOB respectively. The list in Table 2 on page 29 contains entries for the three LOB data types, the corresponding symbolic name, and the default C symbolic name. The LOB symbolic constants can be specified or returned on any of the DB2 CLI functions that take in or return an SQL data type argument (such as `SQLBindParameter()`, `SQLDescribeCol()`).

Since LOB values can be very large, transfer of data using the piecewise sequential method provides by `SQLGetData()` and `SQLPutData()` can be quite time consuming. Applications dealing with such data will often do so in random access segments or via direct file input an output.

There are many cases where an application needs to select a large object value and operate on pieces of it, but does not need or want the entire value to be transferred from the database server into application memory. In these cases, the application can reference an individual LOB value via a large object locator (LOB locator).

A LOB locator is a mechanism that allows an application program to manipulate a large object value in an efficient, random access fashion. A LOB locator is a run time concept: it is not a persistent type and is not stored in the database; it is a mechanism used to refer to a LOB value during a transaction and does not persist beyond the transaction in which it was created. The three LOB locator types each has its own C data type (SQL_C_BLOB_LOCATOR, SQL_C_CLOB_LOCATOR, SQL_C_DBCLOB_LOCATOR). These types are used to enable transfer of LOB locator values to and from the database server.

A LOB locator is a simple token value that represents a single LOB value. A locator is not a reference to a column in a row, rather it is created to reference a large object *value*. There is no operation that could be performed on a locator that would have an effect on the original LOB value stored in the row. An application can retrieve a LOB locator into an application variable (using the `SQLBindCol()` or `SQLGetData()` functions) and can then apply the following DB2 CLI functions to the associated LOB value via the locator:

`SQLGetLength()`
: Gets the length of a string that is represented by a LOB locator.

`SQLGetPosition()`
: Gets the position of a search string within a source string where the

source string is represented by a LOB locator. The search string can also be represented by a LOB locator.

Locators are implicitly allocated by:

- Fetching a bound LOB column to the appropriate C locator type.
- Calling SQLGetSubString() and specifying that the substring be retrieved as a locator.
- Calling SQLGetData() on an unbound LOB column and specifying the appropriate C locator type. The C locator type must match the LOB column type or an error will occur.

LOB locators also provide an efficient method of moving data from one column of a table at the server to another column (of the same or different table) without having to pull the data first into application memory and then sending it back to the server. For example, the following INSERT statement inserts a LOB value that is a concatenation of 2 LOB values as represented by their locators:

```
INSERT INTO lobtable values (CAST ? AS CLOB(4k) || CAST ? AS CLOB(5k))
```

The locator can be explicitly freed before the end of a transaction by executing the FREE LOCATOR statement. The syntax is shown below.

Although this statement cannot be prepared dynamically, DB2 CLI will accept it as a valid statement on SQLPrepare() and SQLExecDirect(). The application uses SQLBindParameter() with the SQL data type argument set to the appropriate SQL and C symbolic data types from Table 2 on page 29.

Alternatively, if the application does require the entire LOB column value, it can request direct file input and output for LOBs. Database queries, updates, and inserts may involve transfer of single LOB column values into and from files. The two DB2 CLI LOB file access functions are:

SQLBindFileToCol()
Binds (associates) a LOB column in a result set with a file name.

SQLBindFileToParam()
Binds (associates) a LOB parameter marker with a file name.

The file name is either the complete path name of the file (which is recommended), or a relative file name. If a relative file name is provided, it is appended to the current path (of the operating environment) of the client process. On execute or fetch, data transfer to and from the file would take

place, similar to bound application variables. A file options argument associated with these 2 functions indicates how the files are to be handled at time of transfer.

Use of SQLBindFileToParam() is more efficient than the sequential input of data segments using SQLPutData() since SQLPutData() essentially puts the input segments into a temporary file and then uses the SQLBindFileToParam() technique to send the LOB data value to the server. Applications should take advantage of SQLBindFileToParam() instead of using SQLPutData().

Refer to "Appendix C. DB2 CLI and ODBC" on page 779 for information on writing generic ODBC applications that use SQL_LONGVARCHAR and SQL_LONGVARBINARY to respectively reference character and binary large object data.

Not all DB2 servers currently have Large Object support, to determine if any of the LOB functions are supported for the currently server, call SQLGetFunctions() with the appropriate function name argument value.

Figure 16 on page 119 shows the retrieval of a character LOB (CLOB).

- The left hand side shows a locator being used to extract a character string from the CLOB, without having to transfer the entire CLOB to an application buffer.

  A LOB locator is fetched, which is then used as an input parameter to search the CLOB for a substring, the substring is then retrieved.
- The right hand side shows how the CLOB can be fetched directly into a file.

  The file is first bound to the CLOB column, and when the row is fetched, the entire CLOB value is transferred directly to a file.

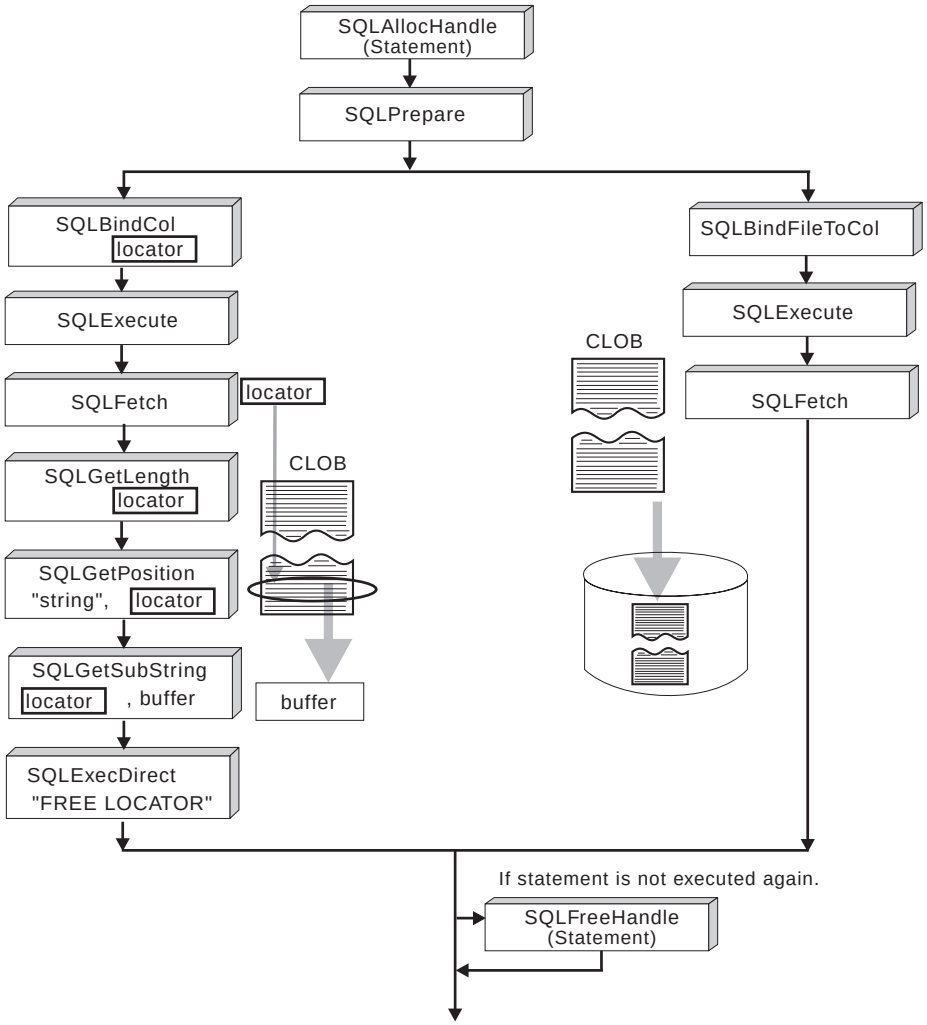


Figure 16. Fetching CLOB Data

LOB Examples

The following example extracts the ″Interests″ section from the Resume CLOB column of the EMP_RESUME table. Only the substring is transferred to the application.

```
/* From CLI sample lookres.c */
/* ... */
    SQLCHAR * stmt2 = "SELECT resume FROM emp_resume "
                      "WHERE empno = ?  AND resume_format = 'ascii'" ;
/* ... */
```

```
/* Get CLOB locator to selected Resume */

rc = SQLBindParameter( hstmt,
                       1,
                       SQL_PARAM_OUTPUT,
                       emp_no.type,
                       SQL_CHAR,
                       emp_no.length,
                       0,
                       emp_no.s,
                       emp_no.length,
                       &emp_no.ind
                     ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

printf( "\n>Enter an employee number:\n" ) ;
gets( ( char * ) emp_no.s ) ;

rc = SQLExecDirect( hstmt, stmt2, SQL_NTS ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol( hstmt,
                 1,
                 SQL_C_CLOB_LOCATOR,
                 &ClobLoc1,
                 0,
                 &pcbValue
               ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLFetch( hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/*
 Search CLOB locator to find "Interests"
 Get substring of resume ( from position of interests to end )
*/

rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &lhstmt ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

/* Get total length */
rc = SQLGetLength( lhstmt,
                   SQL_C_CLOB_LOCATOR,
                   ClobLoc1,
                   &SLength,
                   &Ind ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, lhstmt, rc ) ;

/* Get Starting postion */
rc = SQLGetPosition( lhstmt,
                     SQL_C_CLOB_LOCATOR,
                     ClobLoc1,
                     0,
                     ( SQLCHAR * ) "Interests",
```

```
                    9,
                    1,
                    &Pos1,
                    &Ind
                  ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, lhstmt, rc ) ;

rc = SQLFreeStmt( lhstmt, SQL_CLOSE ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, lhstmt, rc ) ;

buffer = ( SQLCHAR * ) malloc( SLength - Pos1 + 1 ) ;

/* Get just the "Interests" section of the Resume CLOB */
/* ( From Pos1 to end of CLOB ) */
rc = SQLGetSubString( lhstmt,
                      SQL_C_CLOB_LOCATOR,
                      ClobLoc1,
                      Pos1,
                      SLength - Pos1,
                      SQL_C_CHAR,
                      buffer,
                      SLength - Pos1 + 1,
                      &OutLength,
                      &Ind
                    ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, lhstmt, rc ) ;
/* Print Interest section of Employee's resume */
printf( "\nEmployee #: %s\n %s\n", emp_no.s, buffer ) ;
```

Using LOBs in ODBC Applications

Existing ODBC applications use SQL_LONGVARCHAR and SQL_LONGVARBINARY instead of the DB2 BLOB and CLOB data types. By setting the LONGDATACOMPAT keyword in the initialization file, or setting the SQL_ATTR_LONGDATA_COMPAT connection attribute using `SQLSetConnectAttr()`, DB2 CLI will map the ODBC long data types to the DB2 LOB data types.

When this mapping is in effect:

- `SQLGetTypeInfo()` will return CLOB, BLOB and DBCLOB characteristics when called with SQL_LONGVARCHAR, SQL_LONGVARBINARY or SQL_LONGVARGRAPHIC.
- The following functions will return SQL_LONGVARCHAR, SQL_LONGVARBINARY or SQL_LONGVARGRAPHIC when describing CLOB, BLOB or DBCLOB data types:
  - `SQLColumns()`
  - `SQLSpecialColumns()`
  - `SQLDescribeCol()`
  - `SQLColAttribute()`

  - SQLProcedureColumns()
- LONG VARCHAR and LONG VARCHAR FOR BIT DATA will continue to be described as SQL_LONGVARCHAR and SQL_LONGVARBINARY.

The default setting for SQL_ATTR_LONGDATA_COMPAT is SQL_LD_COMPAT_NO, mapping is not in effect.

For more information, refer to "Configuration Keywords" on page 164, and "SQLSetConnectAttr - Set Connection Attributes" on page 618.

With the mapping in effect, ODBC applications can retrieve LOB data by using the SQLGetData(), SQLPutData() and related functions. For more information about inserting and retrieving data in pieces, refer to "Sending/Retrieving Long Data in Pieces" on page 80.

Note: DB2 CLI uses a temporary file when inserting LOB data in pieces. If the data originates in a file, the use of a temporary file can be avoided by using SQLBindFileToParam(). Call SQLGetFunctions() to query if support is provided for SQLBindFileToParam().

Using User Defined Types (UDT)

In addition to the SQL data types (referred to as *base* SQL data types) defined in "Data Types and Data Conversion" on page 28, new distinct types can be defined by the user. These user defined types (UDTs) share its internal representation with an existing type, but is considered to be a separate and incompatible type for most operations. These UDTs are created using the CREATE DISTINCT TYPE SQL statement.

UDTs help provide the strong typing control needed in object oriented programming by ensuring that only those functions and operators explicitly defined on a distinct type can be applied to its instances. Applications continue to work with C data types for application variables, and only need to consider the UDT types when constructing SQL statements.

This means:

- All SQL to C data type conversion rules that apply to the built-in type apply to the UDT.
- The UDT will have the same default C Type as the built-in type.
- SQLDescribeCol() will return the built-in type information. The user defined type name can be obtained by calling SQLColAttribute() with the input descriptor type set to SQL_DESC_DISTINCT_TYPE.

- SQL predicates that involve parameter markers must be explicitly cast to the UDT. This is required since the application can only deal with the built-in types, so before any operation can be performed using the parameter, it must be cast from the C built-in type to the UDT; otherwise an error will occur when the statement is prepared. Refer to "User Defined Types in Predicates" on page 776 for more information.

For complete rules and a description of user defined types(UDT) refer to the *SQL Reference*.

User Defined Type Example

This example shows some UDTs and UDFs being defined, as well as some tables with UDT columns. For an example that inserts rows into a table with UDT columns, refer to "Example" on page 581.

```
/* From CLI sample create.c */
/* ... */
    /* Initialize SQL statement strings */
    SQLCHAR * stmt[] = {

        "CREATE DISTINCT TYPE CNUM AS INTEGER WITH COMPARISONS",

        "CREATE DISTINCT TYPE PUNIT AS CHAR(2) WITH COMPARISONS",

        "CREATE DISTINCT TYPE UPRICE AS DECIMAL(10, 2) "
        "WITH COMPARISONS",

        "CREATE DISTINCT TYPE PRICE AS DECIMAL(10, 2) "
        "WITH COMPARISONS",

        "CREATE FUNCTION PRICE( CHAR(12), PUNIT, char(16) ) "
        "returns char(12) "
        "NOT FENCED EXTERNAL NAME 'order!price' "
        "NOT VARIANT NO SQL LANGUAGE C PARAMETER STYLE DB2SQL "
        "NO EXTERNAL ACTION",

        "CREATE DISTINCT TYPE PNUM AS INTEGER WITH COMPARISONS",

        "CREATE FUNCTION \"+\"(PNUM, INTEGER) RETURNS PNUM "
        "source sysibm.\"+\"(integer, integer)",

        "CREATE FUNCTION MAX(PNUM) RETURNS PNUM "
        "source max(integer)",

        "CREATE DISTINCT TYPE ONUM AS INTEGER WITH COMPARISONS",

        "CREATE TABLE CUSTOMER ( "
        "Cust_Num     CNUM NOT NULL, "
        "First_Name   CHAR(30) NOT NULL, "
        "Last_Name    CHAR(30) NOT NULL, "
        "Street       CHAR(128) WITH DEFAULT, "
```

```
        "City         CHAR(30) WITH DEFAULT, "
        "Prov_State   CHAR(30) WITH DEFAULT, "
        "PZ_Code      CHAR(9) WITH DEFAULT, "
        "Country      CHAR(30) WITH DEFAULT, "
        "Phone_Num    CHAR(20) WITH DEFAULT, "
        "PRIMARY KEY  (Cust_Num) )",

        "CREATE TABLE PRODUCT ( "
        "Prod_Num     PNUM NOT NULL, "
        "Description  VARCHAR(256) NOT NULL, "
        "Price        DECIMAL(10,2) WITH DEFAULT , "
        "Units        PUNIT NOT NULL, "
        "Combo        CHAR(1) WITH DEFAULT, "
        "PRIMARY KEY (Prod_Num), "
        "CHECK (Units in (PUNIT('m'), PUNIT('l'), PUNIT('g'), PUNIT('kg'), "
        "PUNIT(' '))  )",

        "CREATE TABLE PROD_PARTS ( "
        "Prod_Num     PNUM NOT NULL, "
        "Part_Num     PNUM NOT NULL, "
        "Quantity     DECIMAL(14,7), "
        "PRIMARY KEY (Prod_Num, Part_Num), "
        "FOREIGN KEY (Prod_Num) REFERENCES Product, "
        "FOREIGN KEY (Part_Num) REFERENCES Product, "
        "CHECK (Prod_Num <> Part_Num) )",

        "CREATE TABLE ORD_CUST ( "
        "Ord_Num      ONUM NOT NULL, "
        "Cust_Num     CNUM NOT NULL, "
        "Ord_Date     DATE NOT NULL, "
        "PRIMARY KEY (Ord_Num), "
        "FOREIGN KEY (Cust_Num) REFERENCES Customer )",

        "CREATE TABLE ORD_LINE ( "
        "Ord_Num      ONUM NOT NULL, "
        "Prod_Num     PNUM NOT NULL, "
        "Quantity     DECIMAL(14,7), "
        "PRIMARY KEY (Ord_Num, Prod_Num), "
        "FOREIGN KEY (Prod_Num) REFERENCES Product, "
        "FOREIGN KEY (Ord_Num) REFERENCES Ord_Cust )",

        ( char * ) 0,

    } ;

/* ... */

    /* Execute Direct statements */
    i = 0 ;
    while ( stmt[i] != ( char * ) 0 ) {
       printf( ">Executing Statement %ld\n", ( i + 1 ) ) ;
       rc = SQLExecDirect( hstmt, stmt[i], SQL_NTS ) ;
```

```
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
    i++ ;
}
```

Using Stored Procedures

An application can be designed to run in two parts, one on the client and the other on the server. The stored procedure is the part that runs at the database within the same transaction as the application. Stored procedures can be written in either embedded SQL or using the DB2 CLI functions (see "Writing a Stored Procedure in CLI" on page 130). In general, stored procedures have the following advantages:

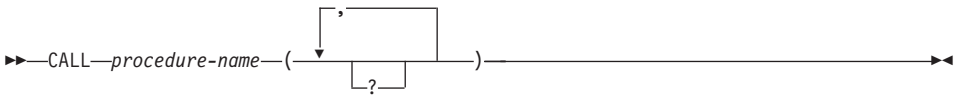
- Avoid network transfer of large amounts of data obtained as part of intermediate results in a long sequence of queries.
- Deployment of client database applications into client/server pieces.

In addition, stored procedures written in embedded static SQL have the following advantages:

- Performance - static SQL is prepared at precompile time and has no run time overhead of access plan (package) generation.
- Encapsulation (information hiding) - users' do not need to know the details about the database objects in order to access them. Static SQL can help enforce this encapsulation.
- Security - users' access privileges are encapsulated within the package(s) associated with the stored procedure(s), so there is no need to grant explicit access to each database object. For example, a user can be granted run access for a stored procedure that selects data from tables for which the user does not have select privilege.

Calling Stored Procedures

Stored procedures are invoked from a DB2 CLI application by passing the following CALL statement syntax to `SQLExecDirect()` or to `SQLPrepare()` followed by `SQLExecute()`.



Note:

Although the CALL statement cannot be prepared dynamically, DB2 CLI accepts the CALL statement as if it could be dynamically prepared.

Stored procedures can also be called using the ODBC vendor escape sequence shown in "Stored Procedure Call Syntax" on page 146.

procedure-name

Specifies a stored procedure name, and it can take one of the following forms:

- *procedure-name*

  The name (with no extension) of the procedure to execute. The procedure invoked is determined as follows.

  1. The *procedure-name* is used both as the name of the stored procedure library and the function name within that library. For example, if *procedure-name* is `proclib`, the DB2 server will load the stored procedure library named `proclib` and execute the function routine `proclib()` within that library.
  2. If the library or function could not be found, the *procedure-name* is used to search the defined procedures (in SYSCAT.PROCEDURES) for a matching procedure. A matching procedure is determined using the steps that follow.
     a. Find the procedures from the catalog (SYSCAT.PROCEDURES) where the PROCNAME matches the *procedure-name* specified and the PROCSCHEMA is a schema name in the function path.
     b. Next, eliminate any of these procedures that do not have the same number of parameters as the number of arguments specified in the CALL statement.
     c. Chose the remaining procedure that is earliest in the function path.
     d. If there are no remaining procedures after step 2, an error is returned (SQLSTATE 42884).

     Once the procedure is selected, DB2 will invoke the procedure defined by the external name.

- *procedure-name!func-name*

  The use of the exclamation sign allows the specification of a library name identified by procedure-name and the function to be executed is given by func-name. This allows similar function routines to be placed in the same stored procedure library.

- /u/db2user/procedure-name!func-name

  The name of the stored procedure library is specified as a full path name. The function to be executed is given by func-name.

For more information regarding the use of the CALL statement and stored procedures, refer to the *SQL Reference* and the *Application Development Guide*.

If the server is DB2 Universal Database Version 2.1 or later, or DB2 for MVS/ESA V4.1 or later, SQLProcedures() can be called to obtain a list of stored procedures available at the database.

Note: For DB2 Universal Database, SQLProcedures() may not return all procedures, and applications can use any valid procedure, regardless of whether it is returned by SQLProcedures(). For more information, refer to "Registering Stored Procedures" and "SQLProcedures - Get List of Procedure Names" on page 603.

The ? in the CALL statement syntax diagram denote parameter markers corresponding to the arguments for a stored procedure. All arguments must be passed using parameter markers; literals, the NULL keyword, and special registers are not allowed. However, literals can be used if the vendor escape call statement is used, ie. the call statement is surrounded by curly braces '{...}'.

The parameter markers in the CALL statement are bound to application variables using SQLBindParameter(). Although stored procedure arguments can be used both for input and output, in order to avoid sending unnecessary data between the client and the server, the application should specify on SQLBindParameter() the parameter type of an input argument to be SQL_PARAM_INPUT and the parameter type of an output argument to be SQL_PARAM_OUTPUT. Those arguments that are both input and output have a parameter type of SQL_PARAM_INPUT_OUTPUT.

If the server is DB2 Universal Database Version 2.1 or later, or DB2 for MVS/ESA V4.1 or later, an application can call SQLProcedureColumns() to determine the type of a parameter in a procedure call. For more information, refer to "Registering Stored Procedures" below and "SQLProcedureColumns - Get Input/Output Parameter Information for A Procedure" on page 594.

Registering Stored Procedures

For DB2 Universal Database, the stored procedure must be registered on the server (in SYSCAT.PROCEDURES and SYSCAT.PROCPARMS) before SQLProcedures() and SQLProcedureColumns() can be invoked; otherwise, these two catalog function calls will return empty result sets. For information on registering stored procedures on the server, see "Appendix H. Pseudo Catalog Table for Stored Procedure Registration" on page 839.

If the stored procedure resides on a DB2 for MVS/ESA V4.1 or later server, the name of the stored procedure must be defined in the

SYSIBM.SYSPROCEDURES catalog table. The pseudo catalog table used by DB2 Universal Database is a derivation and extension of the DB2 for MVS/ESA SYSIBM.SYSPROCEDURES catalog table).

If the stored procedure resides on a DB2 Universal Database for AS/400 V3.1 server, the application must know the actual path and name of the stored procedure ahead of time as there is no real or pseudo catalog table to retrieve information on stored procedures or their argument list.

Handling Stored Procedure Arguments (SQLDA)

Although stored procedures are, in most ways, like any other application, stored procedures written in CLI (and embedded SQL) must give special consideration to the SQLDA structure which contains the stored procedure arguments. The SQLDA structure is described in detail in the *SQL Reference*.

It is important to understand that all data stored in the SQLDA structure is stored as an SQL data type, and must be treated as such by the stored procedure. For example,

- Strings are never null-terminated.
- CHAR types are blank padded.
- VARCHAR and LONG VARCHAR types have both a defined (maximum) length, and an actual length stored as the first two bytes (SQLCHAR structure).
- DECIMAL (or NUMERIC) types are stored in packed decimal format.
- LOB or UDF types cause a *doubled-SQLDA* to be sent.

The suggested approach is for the stored procedure to interpret the SQLDA and move all input arguments to host language variables on entry, and from host language variables to the SQLDA on exit. This allows for SQLDA specific code to be localized within the stored procedure.

Returning Result Sets from Stored Procedures

DB2 CLI provides the ability to retrieve one or more result sets from a stored procedure call, provided the stored procedure has been coded such that one or more cursors, each associated with a query, has been opened and left opened when the stored procedure exits. If more than one cursor is left open, multiple result sets are returned.

Processing within the CLI Application

DB2 CLI applications can retrieve result sets after the execution of a stored procedure that has left cursor(s) open by doing the following:

- Before the stored procedure is called, ensure that there are no open cursors associated with the statement handle.
- Ensure that all cursors previously opened in any previous calls to stored procedures are closed.
- Call the stored procedure.
- The execution of the stored procedure CALL statement effectively causes the opening of the cursor(s) associated with the result set(s).
- Examine any output parameters that have been returned by the stored procedure. For example, the procedure may have been designed so that there is an output parameter that indicates exactly how many result sets have been generated.
- The DB2 CLI application can then use all the normal functions that it has available to process a regular query. If the application does not know the nature of the result set or the number of columns returned, it can call `SQLNumResultCols()`, `SQLDescribeCol()` or `SQLColAttribute()`. Next, it can choose to use any permitted combination of `SQLBindCol()`, `SQLFetch()`, and `SQLGetData()` to obtain the data in the result set.
- When `SQLFetch()` has returned SQL_NO_DATA_FOUND or if the application is done with the current result set, the application can call `SQLMoreResults()` to determine if there are more result sets to retrieve. Calling `SQLMoreResults()` will close the current cursor and advance processing to the next cursor that has been left open by the stored procedure.
- If there is another result set, then `SQLMoreResults()` will return success; otherwise, an SQL_NO_DATA_FOUND is returned.
- Result sets must be processed in serial fashion by the application.

Programming Stored Procedures to Return Result Sets

DB2 Universal Database stored procedures must satisfy the following requirements to return one or more result sets to a CLI application:

- The stored procedure must be run in FENCED mode. If this is not the case then no result sets will be returned, and no error is generated. For more information on fenced and unfenced stored procedures refer to the *Application Development Guide*
- The stored procedure indicates that a result set is to be returned by declaring a cursor on the result set, opening a cursor on the result set (i.e. executing the query), and leaving the cursor open when exiting the stored procedure.
- For every cursor that is left open, a result set is returned to the application.
- If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened in the stored procedure.

- If the stored procedure commits the current transaction then all cursors not declared using the WITH HOLD clause will be closed.
- If the stored procedure rolls back the current transaction then all cursors will be closed.
- If the stored procedure calls `SQLFreeStmt()` with SQL_CLOSE, then the cursor for the current result set is closed and the rows are flushed. Note that this is also the case for all other cursors associated with other result sets generated by this same stored procedure call.
- Only unread rows are passed back. For example, if the result set of a cursor has 500 rows, and 150 of those rows have already been read by the stored procedure at the time the stored procedure terminates, then rows 151 through 500 will be returned to the stored procedure. This can be useful if the stored procedure wishes to filter out some initial rows and not return them to the application.

How Returning a Result Set Differs from Executing a Query Statement

In general, calling a stored procedure that returns a result set is equivalent to executing a query statement. The following restrictions apply:

- Column names are not returned by either `SQLDescribeCol()` or `SQLColAttribute()` for static query statements. In this case, the ordinal position of the column is returned instead.
- The length value of LOB data types is always set to the maximum length, or the value of the LOBMAXCOLUMNSIZE keyword if it is specified.
- All result sets are read-only.
- The cursor cannot be used as a scrollable cursor.
- Schema functions (such as `SQLTables()`) cannot be used to return a result set. If schema functions are used within a stored procedure, all of the cursors for the associated statement handles must be closed before returning, otherwise extraneous result sets may be returned.
- When a query is prepared, result set column information is available before the execute. When a stored procedure is prepared, the result set column information is not available until the CALL statement is executed.

Writing a Stored Procedure in CLI

Although embedded SQL stored procedures provide the most advantages, application developers who have existing DB2 CLI applications may wish to move components of the application to run on the server. In order to minimize the required changes to the code and logic of the application, these components can be implemented as stored procedures, written using DB2 CLI.

Since all the internal information related to a DB2 CLI connection is referenced by the connection handle and since a stored procedure runs under

the same connection and transaction as the client application, it is necessary that a stored procedure written using DB2 CLI make a null SQLConnect() call to associate a connection handle with the underlying connection of the client application. A null SQLConnect() is where the *ServerName*, *UserName*, and *Authentication* argument pointers are all set to NULL and their respective length arguments all set to 0. Of course, in order that an SQLConnect() call can be made at all, the environment and connection handles must already be allocated.

Note: Stored procedures written using Embedded SQL must be precompiled with the DATETIME ISO option in order for DB2 CLI to deal with date-time values correctly.

Stored Procedure Example

The following shows a stored procedure and an example that calls it. (The following example is an input example, see the outcli2.c, outsrv2.c samples for an output example.)

DB2 also includes a number of example programs that demonstrate stored procedures that return multi-row result sets (see the set of example programs that begin with mrsp: mrspcli.c, mrspcli2.c, mrspcli3.sqc, clicall.c, mrspsrv.c and mrspsrv2.sqc).

```
/* From CLI sample inpsrv2.c */
/* ... */
/************************************************************************
*
* PURPOSE: This sample program demonstrates stored procedures,
*          using CLI.  It is rewrite of the inpsrv.sqc embedded SQL
*          stored procedure.
*
*          There are two parts to this program:
*              - the inpcli2 executable (placed on the client)
*              - the inpsrv2 library (placed on the server)
*          CLI stored procedures can be called by either CLI or embbeded
*          applications.
*
*          The inpsrv function will take the information
*          received in the SQLDA to create a table and insert the
*          names of the presidents.
*
*          Refer to the inpcli2.c program for more details on how
*          this program is invoked as the inpsrv2 function
*          in the inpsrv2 library by the EXEC SQL CALL statement.
*
*          The SQL CALL statement will pass in 2 identical SQLDA
*          structures for input and output because all parameters
*          on the CALL statement are assummed to have both the
*          input and output attributes.  However, only changes
*          make to the data and indicator fields in the output SQLDA
*          will be returned to the client program.
```

```
*
* NOTE:    One technique to minimize network flow is to set the
*          variables that returns no output to null on the server program
*          before returning to the client program.
*          This can be achieved by setting the value -128 to the
*          indicator value associated with the data.
*
*          The sqleproc API will call the inpsrv routine stored
*          in the inpsrv library.
*
*          The inpsrv routine will take the information received
*          and create a table called "Presidents" in the "sample"
*          database.  It will then place the values it received in
*          the input SQLDA into the "Presidents" table.
*
*********************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcli1.h>
#include "samputil.h"          /* Header file for CLI sample code */

int SQL_API_FN inpsrv2( struct sqlchar * input_data,
                        struct sqlda   * input_sqlda,
                        struct sqlda   * output_sqlda,
                        struct sqlca   * ca
                      ) {

   /* Declare a local SQLCA */
   struct sqlca sqlca ;

   SQLCHAR table_stmt[80] = "CREATE TABLE " ;
   SQLCHAR insert_stmt[80] = "INSERT INTO " ;
   SQLCHAR insert_data[21] ;
   SQLINTEGER insert_data_ind ;

   /* Delare Miscellanous Variables */
   int cntr ;
   char * table_name ;
   short table_name_length ;
   char * data_item[3] ;
   short data_item_length[3] ;
   int num_of_data = 0 ;

   /* Delare CLI Variables */
   SQLHANDLE henv, hdbc, hstmt ;
   SQLRETURN rc ;

   /*-----------------------------------------------------------------*/
   /* Assign the data from the SQLDA to local variables so that we    */
   /* don't have to refer to the SQLDA structure further.  This will  */
   /* provide better portability to other platforms such as DB2 MVS   */
   /* where they receive the parameter list differently.              */
```

```
/* Note: Strings are not null-terminated in the SQLDA.             */
/*-----------------------------------------------------------------*/

table_name = input_sqlda->sqlvar[0].sqldata ;
table_name_length = input_sqlda->sqlvar[0].sqllen ;
num_of_data = input_sqlda->sqld - 1 ;
for ( cntr = 0; cntr < num_of_data; cntr++ ) {
    data_item[cntr] = input_sqlda->sqlvar[cntr+1].sqldata ;
    data_item_length[cntr] = input_sqlda->sqlvar[cntr+1].sqllen ;
}

/*-----------------------------------------------------------------*/
/* Setup CLI required environment                                  */
/*-----------------------------------------------------------------*/

SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc ) ;

/*-----------------------------------------------------------------*/
/* Issue NULL Connect, since in CLI we need a statement handle     */
/* and thus a connection handle and environment handle.            */
/* A connection is not established, rather the current             */
/* connection from the calling application is used                 */
/*-----------------------------------------------------------------*/
SQLConnect( hdbc, NULL, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS ) ;
SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;

/*-----------------------------------------------------------------*/
/* Create President Table                                          */
/* - For simplicity, we'll ignore any errors from the              */
/*   CREATE TABLE so that you can run this program even when the   */
/*   table already exists due to a previous run.                   */
/*-----------------------------------------------------------------*/

strncat( ( char * ) table_stmt,
         ( char * ) table_name,
         table_name_length
       ) ;
strcat( ( char * ) table_stmt, " (name CHAR(20))" ) ;

SQLExecDirect( hstmt, table_stmt, SQL_NTS ) ;

SQLFreeStmt( hstmt, SQL_RESET_PARAMS ) ;

/*-----------------------------------------------------------------*/
/* Generate and execute a PREPARE for an INSERT statement, and     */
/* then insert the three presidents.                               */
/*-----------------------------------------------------------------*/

strncat( ( char * ) insert_stmt,
         ( char * ) table_name,
         table_name_length
       ) ;
strcat( ( char * ) insert_stmt, "  VALUES (?)" ) ;
```

```
    if ( SQLPrepare(hstmt, insert_stmt, SQL_NTS) != SQL_SUCCESS ) goto ext ;

    /* Bind insert_data to parameter marker */
    SQLBindParameter( hstmt,
                      1,
                      SQL_PARAM_INPUT,
                      SQL_C_CHAR,
                      SQL_CHAR,
                      20,
                      0,
                      insert_data,
                      21,
                      &insert_data_ind
                    ) ;

    for ( cntr = 0; cntr < num_of_data; cntr++ ) {
        strncpy( ( char * ) insert_data,
                 ( char * ) data_item[cntr],
                 data_item_length[cntr]) ;
        insert_data_ind = data_item_length[cntr] ;
        if ( SQLExecute( hstmt ) != SQL_SUCCESS ) goto ext ;
    }

    /*-------------------------------------------------------------------*/
    /* Return to caller                                                  */
    /*   -  Copy the SQLCA                                               */
    /*   -  Update the output SQLDA.  Since there's no output to         */
    /*      return, we are setting the indicator values to -128 to       */
    /*      return only a null value.                                    */
    /*   -  Commit or Rollback the inserts.                              */
    /*-------------------------------------------------------------------*/

ext:

     rc = SQLGetSQLCA( henv, hdbc, hstmt, &sqlca ) ;
     if ( rc != SQL_SUCCESS ) printf( "RC = %d\n", rc ) ;
     memcpy( ca, &sqlca, sizeof( sqlca ) ) ;
     if ( output_sqlda != NULL ) {
        for ( cntr = 0; cntr < output_sqlda->sqld; cntr++ ) {
            if ( output_sqlda->sqlvar[cntr].sqlind != NULL )
               *( output_sqlda->sqlvar[cntr].sqlind ) = -128 ;
        }
     }

     rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
     CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

     rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
     CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

     printf( ">Disconnecting .....\n" ) ;
     rc = SQLDisconnect( hdbc ) ;
     CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

     rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc ) ;
```

```
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

    rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
    if ( rc != SQL_SUCCESS )
        return( terminate( henv, rc ) ) ;

    return( SQL_SUCCESS ) ;

}
/* From CLI sample inpcli2.c */
/* ... */
    SQLCHAR * stmt = "CALL inpsrv2(?, ?, ?, ?)" ;
/* ... */

    rc = SQLPrepare( hstmt, stmt, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindParameter( hstmt,
                           1,
                           SQL_PARAM_INPUT,
                           SQL_C_CHAR,
                           SQL_CHAR,
                           9,
                           0,
                           Tab_Name,
                           10,
                           NULL
                         ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindParameter( hstmt,
                           2,
                           SQL_PARAM_INPUT,
                           SQL_C_CHAR,
                           SQL_CHAR,
                           10,
                           0,
                           Pres_Name[0],
                           11,
                           NULL
                         ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindParameter( hstmt,
                           3,
                           SQL_PARAM_INPUT,
                           SQL_C_CHAR,
                           SQL_CHAR,
                           10,
                           0,
                           Pres_Name[1],
                           11,
                           NULL
                         ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

```
rc = SQLBindParameter( hstmt,
                       4,
                       SQL_PARAM_INPUT,
                       SQL_C_CHAR,
                       SQL_CHAR,
                       10,
                       0,
                       Pres_Name[2],
                       11,
                       NULL
                     ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLExecute( hstmt ) ;
/* Ignore Warnings */
if ( rc != SQL_SUCCESS_WITH_INFO )
   CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

Mixing Embedded SQL and DB2 CLI

It is possible, and sometimes desirable, for an application to use DB2 CLI in conjunction with embedded static SQL. Consider the scenario where the application developer wishes to take advantage of the ease of use provided by the DB2 CLI catalog functions and maximize the portion of the application's processing where performance is critical. In order to mix the use of DB2 CLI and embedded SQL, the application must comply to the following rules:

- All connection management and transaction management must be performed completely using either DB2 CLI or embedded SQL. Either the DB2 CLI application performs all the Connects and Commits/Rollback and calls functions written using embedded SQL; or an embedded SQL application performs all the Connects and Commits/Rollback and calls functions written in DB2 CLI which use a null connection (see "Writing a Stored Procedure in CLI" on page 130 for details on null connections).
- Query statement processing must not and cannot straddle across DB2 CLI and embedded SQL interfaces for the same statement; for example, the application cannot open a cursor in an embedded SQL routine, and then call the DB2 CLI `SQLFetch()` function to retrieve row data.

Since DB2 CLI permits multiple connections, the `SQLSetConnection()` function must be called prior to making a function call to a routine written in embedded SQL. This allows the application to explicitly specify the connection under which the embedded SQL routine should perform its processing.

If the DB2 CLI application is multithreaded and also makes embedded SQL calls or DB2 API calls, then each thread must have a DB2 context. See "Writing Multi-Threaded Applications" on page 46 for complete details.

Mixed Embedded SQL and DB2 CLI Example

The following example demonstrates an application that connects to two data sources, and executes both embedded SQL and dynamic SQL using DB2 CLI.

```
/* From CLI sample mixed.sqc */
/* ... */
    /* allocate an environment handle   */
    SQLAllocEnv(&henv);

    /* Connect to first data source */
    DBconnect(henv, &hdbc[0]);

    /* Connect to second data source */
    DBconnect(henv, &hdbc[1]);

    /*********   Start Processing Step  ************************/
    /* NOTE: at this point there are two active connections */

    /* set current connection to the first database */
    if ( (rc = SQLSetConnection(hdbc[0])) != SQL_SUCCESS )
         printf("Error setting connection 1\n");

    /* call function that contains embedded SQL */
    if ((rc = Create_Tab() ) != 0)
         printf("Error Creating Table on 1st connection, RC=%d\n", rc);

   /*  Commit transation on connection 1 */
    SQLTransact(henv, hdbc[0], SQL_COMMIT);

    /* set current connection to the second database */
    if ( (rc = SQLSetConnection(hdbc[1])) != SQL_SUCCESS )
         printf("Error setting connection 2\n");

    /* call function that contains embedded SQL */
    if ((rc = Create_Tab() ) != 0)
         printf("Error Creating Table on 2nd connection, RC=%d\n", rc);

   /*  Commit transation on connection 2 */
    SQLTransact(henv, hdbc[1], SQL_COMMIT);

    /* Pause to allow the existance of the tables to be verified. */
    printf("Tables created, hit Return to continue\n");
    getchar();

    SQLSetConnection(hdbc[0]);
    if (( rc = Drop_Tab() ) != 0)
        printf("Error dropping Table on 1st connection, RC=%d\n", rc);

   /*  Commit transation on connection 1 */
```

```
    SQLTransact(henv, hdbc[0], SQL_COMMIT);

    SQLSetConnection(hdbc[1]);
    if (( rc = Drop_Tab() ) != 0)
        printf("Error dropping Table on 2nd connection, RC=%d\n", rc);

   /*  Commit transation on connection 2 */
    SQLTransact(henv, hdbc[1], SQL_COMMIT);

    printf("Tables dropped\n");

    /*********   End Processing Step  **************************/
/* ... */
/*************   Embedded SQL Functions  ******************************
** This would normally be a seperate file to avoid having to           *
** keep precompiling the embedded file in order to compile the DB2 CLI *
** section.                                                            *
**********************************************************************/

#include "sql.h"
#include "sqlenv.h"

EXEC SQL INCLUDE SQLCA;

int
Create_Tab( )
{

   EXEC SQL CREATE TABLE mixedup
            (ID INTEGER, NAME CHAR(10));

   return( SQLCODE);
}

int
Drop_Tab( )
{
   EXEC SQL DROP TABLE mixedup;

   return( SQLCODE);
}
```

Asynchronous Execution of CLI

DB2 CLI can run a subset of functions asynchronously; the DB2 CLI driver returns control to the application after calling the function, but before that function has finished executing. The function returns SQL_STILL_EXECUTING each time it is called until it is finished running, at which point it returns a different value (SQL_SUCCESS for example).

Asynchronous execution is only beneficial on single-threaded operating systems. Applications that run on multithreaded operating systems should execute functions on separate threads instead.

Asynchronous execution is possible for those functions that normally send a request to the server and then wait for a response. Rather than waiting, a function executing asynchronously returns control to the application. The application can then perform other tasks, or return control to the operating system, and use an interrupt to repeatedly poll the function until a return code other than SQL_STILL_EXECUTING is returned.

Typical Asynchronous Application

Each application that will will run functions asynchronously must complete the following steps in addition to the normal CLI steps, in the following order:

1. Set Up the Environment

To ensure that functions can be called asynchronously, the application should call SQLGetInfo() with an option of SQL_ASYNC_MODE.

```
/* See what type of Asynchronous support is available. */
rc = SQLGetInfo( hdbc, /* Connection handle */
                 SQL_ASYNC_MODE, /* Query the support available */
                 &ubuffer, /* Store the result in this variable */
                 4,
                 &outlen);
```

The call to SQLGetInfo() will return one of the following values:

SQL_AM_STATEMENT - Statement Level
Indicates that asynchronous execution can be turned on or off on a statement level.

Statement level asynchronous execution is set using the statement attribute SQL_ATTR_ASYNC_ENABLE. An application can have at most 1 active function running in asynchronous mode on any one connection. It should be set to SQL_ASYNC_ENABLE_ON using SQLSetStmtAttr().

```
/* Set statement level asynchronous execution on */
rc = SQLSetStmtAttr( hstmt, /* Statement handle */
                     SQL_ATTR_ASYNC_ENABLE,
                     (SQLPOINTER) SQL_ASYNC_ENABLE_ON,
                     0);
```

SQL_AM_CONNECTION - Connection Level
DB2 Universal Database supports SQL_AM_STATEMENT, but

SQL_AM_CONNECTION may be returned by other datasources. It indicates that all statements on a connection must execute in the same way.

Connection level asynchronous execution is set using the connection attribute SQL_ATTR_ASYNC_ENABLE. It should be set to SQL_ASYNC_ENABLE_ON using SQLSetConnectAttr().

All statements already allocated, as well as future statement handles allocated on this connection will be enabled for asynchronous execution.

SQL_AM_NONE - Asynchronous execution not supported

This will be returned for one of two reasons:

1. The datasource itself does not support asynchronous execution.
2. The DB2 CLI/ODBC configuration keyword ASYNCENABLE has been specifically set to disable asynchronous execution. See "ASYNCENABLE" on page 167 for more details.

In either case the functions will be executed synchronously. If the application does call SQLSetStmtAttr() or SQLSetConnectAttr() to turn on asynchronous execution, the call will return an SQLSTATE of 01S02 (option value changed).

2. Call a Function that Supports Asynchronous Execution

When the application calls a function that can be run asynchronously one of two things can take place.

- If the function will not benefit from being run asynchronously, DB2 CLI can decide to run it synchronously and return the normal return code (other than SQL_STILL_EXECUTING).

  In this case the application runs as it would if the asynchronous mode had not been enabled.
- DB2 CLI will perform some minimal processing (such as checking the arguments for errors), then pass the statement on to the server. Once this quick processing is complete a return code of SQL_STILL_EXECUTING is returned to the application.

See the SQL_ATTR_ASYNC_ENABLE statement attribute in the SQLSetStmtAttr() function for a list of functions that can be executed asynchronously.

The following example demonstrates a common while loop that takes both possible outcomes into account:

```
while ( (rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS) ) == SQL_STILL_EXECUTING)
{
    /* Other processing can be performed here, between each call to
```

```
     * see if SQLExecDirect() has finished running asynchronously.
     * This section will never run if CLI runs the function
     * synchronously.
     */
}
/* The application continues at this point when SQLExecDirect() */
/* has finished running. */
```

3. Poll Asynchronous Function While Calling Others

The application determines whether the function has completed by calling it repeatedly with the same arguments it used to call the function the first time. A return code of SQL_STILL_EXECUTING indicates it is not yet finished, any other value indicates it has completed. The value other than SQL_STILL_EXECUTING is the same return code it would have returned if it had executed synchronously.

Functions that can be called during Asynchronous execution

The following functions can be called while a function is being executed asynchronously. Any other function will return an SQLSTATE of HY010 (Function sequence error).

- any function on any other statement within the same connection
- any function on any connection other than the one associated with the asynchronous statement
- `SQLCancel()` on the asynchronous statement to stop it (see "5. Cancelling the Asynchronous Function Call" on page 142)
- `SQLGetDiagField()` and `SQLGetDiagRec()` on the asynchronous statement or connection to get a header diagnostic field but not a record diagnostic field (see "4. Diagnostic Information while Running")
- `SQLAllocHandle()` on the connection associated with the asynchronous statement, to allocate a statement handle

4. Diagnostic Information while Running

The following values are returned when `SQLGetDiagField()` is called on a statement handle that has an asynchronous function executing:

- the values of SQL_DIAG_CURSOR_ROW_COUNT, SQL_DIAG_DYNAMIC_FUNCTION, SQL_DIAG_DYNAMIC_FUNCTION_CODE, and SQL_DIAG_ROW_COUNT header fields are undefined
- SQL_DIAG_NUMBER header field returns 0
- SQL_DIAG_RETURN_CODE header field returns SQL_STILL_EXECUTING
- all record fields return SQL_NO_DATA

SQLGetDiagRec() always returns SQL_NO_DATA when it is called on a statement handle that has an asynchronous function executing.

5. Cancelling the Asynchronous Function Call

The application can issue a request to cancel any function that is running asynchronously by calling SQLCancel(). There are cases, however, where this request will not be carried out (if the function has already finished, for example).

The return code from the SQLCancel() call indicates whether the cancel request was received, not whether the execution of the asynchronous function was stopped.

The only way to tell if the function was canceled is to call it again, using the original arguments.

- If the cancel was successful, the function will return SQL_ERROR and an SQLSTATE of HY008 (Operation canceled).
- If the cancel was not successful, the function will either return a value other than SQL_ERROR, or will return SQL_ERROR and an SQLSTATE other than HY008.

Sample Asynchronous Application

The following CLI sample, async.c, demonstrates a simple application that runs SQLExecDirect() asynchronously. It is based on the CLI sample program fetch.c.

```
/* CLI sample async.c */
/* ... */
    /* Make the result from SQLGetInfo() more meaningful by mapping */
    /* the returned value to the string. */
    static char ASYNCMODE[][19] =  {  "SQL_AM_NONE",
                                      "SQL_AM_CONNECTION",
                                      "SQL_AM_STATEMENT" };
/* ... */
     * See what type of Asynchronous support is available,
     * and whether or not the CLI/ODBC configuration keyword ASYNCENABLE
     * is set on or off.
     */
    rc = SQLGetInfo( hdbc, /* Connection handle */
                     SQL_ASYNC_MODE, /* Query the support available */
                     &ubuffer, /* Store the result in this variable */
                     4,
                     &outlen);
    CHECK_STMT(hstmt, rc);

    printf("SQL_ASYNC_MODE value from SQLGetInfo() is %s.\n\n",
            ASYNCMODE[ubuffer]);

    if (ubuffer == SQL_AM_NONE ) { /* Async not supported */
```

```
    printf("Asynchronous execution is not supported by this datasource\n");
    printf("or has been turned off by the CLI/ODBC configuration keyword\n");
    printf("ASYNCENABLE. The application will continue, but\n");
    printf("SQLExecDirect() will not be run asynchronously.\n\n");

    /* There is no need to set the SQLSetStmtAttr() option */
 } else {

    /* Set statement level asynchronous execution on */
    rc = SQLSetStmtAttr(
              hstmt,
              SQL_ATTR_ASYNC_ENABLE,
              (SQLPOINTER) SQL_ASYNC_ENABLE_ON,
              0);
    CHECK_STMT(hstmt, rc);
 }


 /* The while loop is new for the asynchronous sample, the */
 /* SQLExecDirect() call remains the same. */
 while ((rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS) )
          == SQL_STILL_EXECUTING) {
    printf("   ...SQLExecDirect() still executing asynchronously...\n");
    /* Other processing can be performed here, between each call
     * to see if SQLExecDirect() has finished running asynchronously.
     * This section will never run if CLI runs the function
     * synchronously.
     */
}

 CHECK_STMT(hstmt, rc);

 rc = SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) deptname.s, 15,
                 &deptname.ind);
 CHECK_STMT(hstmt, rc);

 rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) location.s, 15,
                 &location.ind);
 CHECK_STMT(hstmt, rc);

 printf("Departments in Eastern division:\n");
 printf("DEPTNAME       Location\n");
 printf("-------------- -------------\n");

 while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
     printf("%-14.14s %-14.14s \n", deptname.s, location.s);
 }
 if (rc != SQL_NO_DATA_FOUND)
     check_error(henv, hdbc, hstmt, rc, __LINE__, __FILE__);

 rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
 CHECK_STMT(hstmt, rc);

 rc = SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT);
 CHECK_DBC(hdbc, rc);
```

```
    printf("Disconnecting .....\n");
    rc = SQLDisconnect(hdbc);
    CHECK_DBC(hdbc, rc);

    rc = SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    CHECK_DBC(hdbc, rc);

    rc = SQLFreeHandle(SQL_HANDLE_ENV, henv);
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));

}                               /* end main */
```

Using Vendor Escape Clauses

The X/Open SQL CAE specification defined an **escape clause** as: "a syntactic mechanism for vendor-specific SQL extensions to be implemented in the framework of standardized SQL". Both DB2 CLI and ODBC support vendor escape clauses as defined by X/Open.

Currently, escape clauses are used extensively by ODBC to define SQL extensions. DB2 CLI translates the ODBC extensions into the correct DB2 syntax. The `SQLNativeSql()` function can be used to display the resulting syntax.

If an application is only going to access DB2 data sources, then there is no reason to use the escape clauses. If an application is going to access other data sources that offer the same support, but uses different syntax, then the escape clauses increase the portability of the application.

DB2 CLI used both the standard and shorthand syntax for escape clauses. The standard syntax has been deprecated (although DB2 CLI still supports it). An escape clause using the standard syntax took the form:

```
--(*vendor(vendor-identifier),
      product(product-identifier) extended SQL text*)--
```

Applications should now only use the shorthand syntax, as described below, to remain current with the latest ODBC standards.

Escape Clause Syntax

The format of an escape clause definition is:

```
{ extended SQL text }
```

to define the following SQL extensions:

- Extended date, time, timestamp data

- Outer join
- LIKE predicate
- Call stored procedure
- Extended scalar functions
  - Numeric functions
  - String functions
  - System functions

ODBC Date, Time, Timestamp Data

The ODBC escape clauses for date, time, and timestamp data are:

```
{d 'value'}
{t 'value'}
{ts 'value'}
```

d indicates *value* is a date in the *yyyy-mm-dd* format,

t indicates *value* is a time in the *hh:mm:ss* format

ts indicates *value* is a timestamp in the *yyyy-mm-dd hh:mm:ss[.f...]* format.

For example, the following statement can be used to issue a query against the **EMPLOYEE** table:

```
SELECT * FROM EMPLOYEE WHERE HIREDATE={d '1994-03-29'}
```

DB2 CLI will translate the above statement to a DB2 format. `SQLNativeSql()` can be used to return the translated statement.

The ODBC escape clauses for date, time, and timestamp literals can be used in input parameters with a C data type of SQL_C_CHAR.

ODBC Outer Join Syntax

The ODBC escape clause for outer join is:

```
{oj outer-join}
```

where *outer join* is

```
table-name  {LEFT | RIGHT | FULL} OUTER JOIN
            {table-name | outer-join}
            ON search-condition
```

For example, DB2 CLI will translate the following statement:

```
SELECT * FROM {oj T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3}
    WHERE T1.C2>20
```

to IBM's format, which corresponds to the SQL92 outer join syntax.

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3 WHERE T1.C2>20
```

Note: Not all DB2 servers support outer join. To determine if the current server supports outer joins, call `SQLGetInfo()` with the SQL_SQL92_RELATIONAL_JOIN_OPERATORS and SQL_OJ_CAPABILITIES options.

LIKE Predicate Escape Clauses

In a SQL LIKE predicate, the metacharacter % matches zero or more of any character and the metacharacter _ matches any one character. The ESCAPE clause allows the definition of patterns intended to match values that contain the actual percent and underscore characters by preceding them with an escape character. The escape clause ODBC uses to define the LIKE predicate escape character is:

{escape '*escape-character***'}**

where *escape-character* is any character supported by the DB2 rules governing the use of the ESCAPE clause.

Applications that are not concerned about portability across different vendor DBMS products should pass the ESCAPE clause directly to the data source. To determine when LIKE predicate escape characters are supported by a particular DB2 data source, an application should call `SQLGetInfo()` with the SQL_LIKE_ESCAPE_CLAUSE information type.

Stored Procedure Call Syntax

The ODBC escape clause for calling a stored procedure is:

{[?=]call *procedure-name[([parameter][,[parameter]]...)]***}**

- *procedure-name*. specifies the name of a procedure stored at the data source
- *parameter* specifies a procedure parameter.

A procedure may have zero or more parameters. (The square brackets ([ ]) indicate optional arguments.)

ODBC species the optional parameter **?=** to represent the procedure's return value, which, if present, will be stored in the location specified by the first parameter marker as defined via `SQLBindParameter()`. DB2 CLI will return the SQLCODE as the procedure's return value if **?=** is present in the escape clause. If **?=** is not present, then the application can retrieve the SQLCA by using the `SQLGetSQLCA()` function. Unlike ODBC, DB2 CLI does not support literals as procedure arguments, parameter markers must be used.

For more information about stored procedures, refer to "Using Stored Procedures" on page 125 or the *Application Development Guide*.

For example, DB2 CLI will translate the following statement:

```
{CALL NETB94(?,?,?)}
```

To an internal CALL statement format:

```
CALL NEBT94(?, ?, ?)
```

ODBC Scalar Functions

Scalar functions such as string length, substring, or trim can be used on columns of a result sets and on columns that restrict rows of a result set. The ODBC escape clauses for scalar functions is:

{fn *scalar-function***}**

Where, *scalar-function* can be any function listed in "Appendix D. Extended Scalar Functions" on page 785.

For example, DB2 CLI will translate of the following statement:

```
SELECT {fn CONCAT(FIRSTNAME,LASTNAME)} FROM EMPLOYEE
```

to:

```
SELECT FIRSTNAME CONCAT LASTNAME FROM EMPLOYEE
```

`SQLNativeSql()` can be called to obtain the translated SQL statement.

To determine which scalar functions are supported by the current server referenced by a specific connection handle, call `SQLGetInfo()` with the SQL_NUMERIC_FUNCTIONS, SQL_STRING_FUNCTIONS, SQL_SYSTEM_FUNCTIONS, and SQL_TIMEDATE_FUNCTIONS options.

Chapter 4. Configuring CLI/ODBC and Running Sample Applications

The DB2 CLI runtime environment is included with any of the DB2 Client Application Enabler products. Development support for each platform is provided by the corresponding DB2 Software Developer's Kit (DB2 SDK) which is part of the separately orderable DB2 Application Development Kit product.

For example, OS/2 applications are developed using DB2 SDK for OS/2, and can run against any DB2 server using DB2 Client Application Enabler for OS/2.

Setting up the DB2 CLI Runtime Environment

Runtime support for DB2 CLI applications is contained in all DB2 Universal Database products, including the DB2 Client Application Enabler and the DB2 SDK. This section describes the general setup required; see also "Platform Specific Details for CLI/ODBC Access" on page 151.

In order for a DB2 CLI application to successfully access a DB2 database:

1. The database (and node if the database is remote) must be cataloged. Use either the command line processor or (if applicable) the DB2 administration tool for your platform.
2. The DB2 CLI bind files must be bound to the database.

   DB2 CLI will auto-bind on the first access to the database, provided the user has the appropriate authorization. The database administrator may need to perform the first connect, or explicitly bind the files. See "How to Bind the DB2 CLI/ODBC Driver to the Database" on page 157 for more information.
3. Optionally the DB2 CLI/ODBC Configuration Keywords can be set.

   See "Platform Specific Details for CLI/ODBC Access" on page 151 for more information on how to do this using the tools available on your platform, or "How to Set CLI/ODBC Configuration Keywords" on page 158 for details on doing this manually.

Running CLI/ODBC Programs

The DB2 Call Level Interface (CLI) run-time environment and the DB2 CLI/ODBC driver are included with DB2 clients as optional components during install.

This support enables applications developed using ODBC and DB2 CLI APIs to work with any DB2 server. DB2 CLI application development support is provided by the DB2 Software Developer's Kit (DB2 SDK) which is packaged with your DB2 server.

Before DB2 CLI or ODBC applications can access DB2, the DB2 CLI packages must be bound on the server. Although this will occur automatically on the first connection if the user has the required authority to bind the packages, it is recommended that the administrator do this first with each version of the client on each platform that will access the server. See "How to Bind the DB2 CLI/ODBC Driver to the Database" on page 157 for specific details.

The following general steps are required on the client system to give DB2 CLI and ODBC applications access to DB2 databases. These instructions assume that you have successfully connected to DB2 using a valid user ID and password. Depending on the platform many of these steps are automatic. For complete details, see the section that deals specifically with your platform.

Step 1. Use the Client Configuration Assistant (CCA) to add the database (if you have separate client and server machines) so that its instances and databases can be made known to the Control Center, then add the instances and databases for that system. (Your local system is represented by **Local** icon.) If you do not have access to this program you can use the **catalog** command in the command line processor.

Step 2. On all platforms other than OS/2 the DB2 CLI/ODBC driver is an optional component during the DB2 client install. Be sure it is selected at that point. On OS/2 you must use the **Install ODBC Driver** icon to install both the DB2 CLI/ODBC driver and the ODBC driver manager.

Step 3. To access the DB2 database from ODBC:

a. The ODBC Driver Manager (From Microsoft or other vendor) must already be installed (this is done by default during the installation of DB2 only on 32-bit Windows systems).

b. The DB2 databases must be registered as ODBC data sources. The ODBC driver manager does not read the DB2 catalog information; instead it references its own list of data sources.

c. If a DB2 table does not have a unique index then many ODBC applications will open it as read-only. A unique index should be created for each DB2 table that is to be updated by an ODBC

application. Refer to the **CREATE INDEX** statement in the *SQL Reference*. Using the Control Center you would alter the settings of the table, then select the **Primary Key** tab and move one or more columns from the available columns list over to the primary key columns list. Any column you select as part of the primary key must be defined as NOT NULL.

Step 4. If necessary, you can set various CLI/ODBC Configuration Keywords to modify the behavior of DB2 CLI/ODBC and the applications using it.

If you followed the above steps to install ODBC support, and added DB2 databases as ODBC data sources, your ODBC applications will now be able to access them.

After the platform specific instructions there are further details on the following topics:

- "How to Bind the DB2 CLI/ODBC Driver to the Database" on page 157
- "How to Set CLI/ODBC Configuration Keywords" on page 158
- "Configuring db2cli.ini" on page 159

Platform Specific Details for CLI/ODBC Access

The platform specific details on how to give DB2 CLI and ODBC applications access to DB2 are divided into the following categories:

- "Windows 32-bit operating systems Client Access to DB2 using CLI/ODBC"
- "OS/2 Client Access to DB2 using CLI/ODBC" on page 153
- "UNIX Client Access to DB2 using CLI/ODBC" on page 155

Windows 32-bit operating systems Client Access to DB2 using CLI/ODBC

Before DB2 CLI and ODBC applications can successfully access a DB2 database from a Windows client, perform the following steps on the client system:

Step 1. The DB2 database (and node if the database is remote) must be cataloged. To do so, use the CCA (or the command line processor).

For more information refer to the on-line help in the CCA (or the **CATALOG DATABASE** and **CATALOG NODE** commands in the *Command Reference*).

Step 2. Verify that the Microsoft ODBC Driver Manager and the DB2 CLI/ODBC driver are installed. On Windows 32-bit operating systems they are both installed with DB2 unless the ODBC component is manually unselected during the install.

To verify that they both exist on the machine:

a. Run the Microsoft ODBC Administrator from the icon in the Control Panel, or issue the appropriate command from the command line: **odbcad32.exe.**
b. Click on the **ODBC Drivers** tab.
c. Verify that ″IBM DB2 ODBC DRIVER″ is shown in the list.

If either the Microsoft ODBC Driver Manager or the IBM DB2 CLI/ODBC driver is not installed, then rerun the DB2 install and select the ODBC component on Windows 32-bit operating systems.

Step 3. Register the DB2 database with the ODBC driver manager as a *data source*. On Windows 32-bit operating systems you can make the data source available to all users of the system (a system data source), or only the current user (a user data source). Use either of these methods to add the data source:

- Using the CCA:
  a. Select the DB2 database alias that you want to add as a data source.
  b. Click on the **Properties** push button. The Database Properties window opens.
  c. Select the **Register this database for ODBC** check box.
  d. On Windows 32-bit operating systems you can use the radio buttons to add the data source as either a user or system data source.
- Using the Microsoft **32-bit ODBC Administration tool**, which you can access from the icon in the Control Panel or by running **odbcad32.exe** from the command line:
  a. On Windows 32-bit operating systems the list of user data sources appears by default. If you want to add a system data source click on the **System DSN** button, or the **System DSN** tab (depending on the platform).
  b. Click on the **Add** push button.
  c. Double-click on the IBM DB2 ODBC Driver in the list.
  d. Select the DB2 database to add and click on **OK**.
- On Windows 32-bit operating systems there is a command that can be issued in the command line processor to register the DB2 database with the ODBC driver manager as a data source. An administrator could create a command line processor script to register the required databases. This script could then be run on all of the machines that require access to the DB2 databases through ODBC.

The *Command Reference* contains more information on the CATALOG command:

```
CATALOG [ user | system ] ODBC DATA SOURCE
```

Step 4. Configure the DB2 CLI/ODBC driver using the CCA: (Optional)

a. Select the DB2 database alias you want to configure.

b. Click on the **Properties** push button. The Database Properties window opens.

c. Click on the **Settings** push button. The CLI/ODBC Settings window opens.

d. Click on the **Advanced** push button. You can set the configuration keywords in the window that opens. These keywords are associated with the database *alias name*, and affect all DB2 CLI/ODBC applications that access the database. The online help explains all of the keywords, as does "Configuration Keywords" on page 164.

For information on manually editing this file (db2cli.ini), see "Configuring db2cli.ini" on page 159.

Step 5. If you have installed ODBC access (as described above), you can now access DB2 data using ODBC applications. Start the ODBC application and go to the Open window. Select the **ODBC databases** file type. The DB2 databases that you added as ODBC data sources will be selectable from the list. Many ODBC applications will open the table as read-only unless a unique index exists.

OS/2 Client Access to DB2 using CLI/ODBC

Before DB2 CLI and ODBC applications can successfully access a DB2 database from an OS/2 client, perform the following steps on the client system:

1. The DB2 database (and node if the database is remote) must be cataloged. To do so, use the CCA (or the command line processor).

   For more information see the on-line help in the CCA (or the **CATALOG DATABASE** and **CATALOG NODE** commands in the *Command Reference*). (or the **CATALOG DATABASE** and **CATALOG NODE** commands in the *Command Reference*).

2. If you are using ODBC applications to access DB2 data, perform the following steps. (If you are using only CLI applications, skip this step and go to the next step.)

   a. Check that there is an ODBC Driver Manager installed. The ODBC Driver Manager is not installed with DB2; we suggest you use the Driver Manager that was shipped with your ODBC application. Also ensure that the DB2 CLI/ODBC driver is installed:

1) Run the ODBC Administration tool as described in its documentation. This is usually done in one of two ways:
   - Double-click on the **ODBC** Folder in OS/2, and double-click on the **ODBC Administrator** icon.
   - Run **odbcadm.exe** from the command line.

   The Data Sources window opens.
2) Click on the **Drivers** push button. The Drivers window opens.
3) Verify that ″IBM DB2 ODBC DRIVER″ is shown in the list.

If the ODBC Driver Manager is not installed then follow the installation instructions that came with your ODBC application. If the IBM DB2 CLI/ODBC driver is not installed then double-click on the **Install ODBC Driver** icon in the DB2 folder to install the DB2 CLI/ODBC driver.

b. Register the DB2 database with the ODBC driver manager as a *data source* using either of these methods:
   - Using the CCA:
     1) Select the DB2 database alias that you want to add as a data source.
     2) Click on the **Properties** push button.
     3) Select the **Register this database for ODBC** check box.
   - Using the ODBC Driver Manager:
     1) Run the ODBC Driver Manager, as described in its documentation. This is usually done in one of two ways:
        – Double-click on the **ODBC** Folder in OS/2, and double-click on the **ODBC Administrator** icon.
        – Run **odbcadm.exe** from the command line.
     2) Click on the **Add** push button from the Data Sources window. The Add Data Source Window opens.
     3) Double-click on the IBM DB2 ODBC DRIVER in the list.
     4) Select the DB2 database to add and click on **OK**.

3. Configure the DB2 CLI/ODBC driver using the CCA: (Optional)
   a. Select the DB2 database alias you want to configure.
   b. Click on the **Properties** push button. The Database Properties window opens.
   c. Click on the **Settings** push button. The CLI/ODBC Settings window opens.
   d. Click on the **Advanced** push button. You can set the configuration keywords in the window that appears. These keywords are associated with the database *alias name*, and affect all DB2 CLI/ODBC

applications that access the database. The online help explains all of the keywords, as does “Configuration Keywords” on page 164.

For information on manually editing this file (db2cli.ini), see “Configuring db2cli.ini” on page 159.

4. If you have installed ODBC access (as described above), you can now access DB2 data using ODBC applications. Start the ODBC application and go to the Open window. Select the **ODBC databases** file type. The DB2 databases that you added as ODBC data sources will be selectable from the list. Many ODBC applications will open the table as read-only unless a unique index exists.

UNIX Client Access to DB2 using CLI/ODBC

Before DB2 CLI and ODBC applications can successfully access a DB2 database from a UNIX client, perform the following steps on the client system:

1. The DB2 database (and node if the database is remote) must be cataloged. To do so, use the command line processor.

   For more information see the **CATALOG DATABASE** and **CATALOG NODE** commands in the *Command Reference*.
2. The DB2 CLI/ODBC driver is an optional component during the DB2 client install. Be sure it is selected at that point.
3. If you are using ODBC applications to access DB2 data, perform the following steps. (If you are using only CLI applications, skip this step and go to the next step.)
   a. When using an ODBC application you must ensure that an ODBC Driver Manager is installed and that each user that will use ODBC has access to it. DB2 does not install an ODBC Driver Manager, you must use the ODBC Driver Manager that was supplied with your ODBC client application or ODBC SDK in order to access DB2 data using that application.
   b. The Driver Manager uses two initialization files.

      odbcinst.ini ODBC Driver Manager’s configuration file indicating which database drivers are installed. Each user that will use ODBC must have access to this file.

      .odbc.ini End-user’s data source configuration. Each user ID has a separate copy of this file in their home directory. Note that the file starts with a dot.

      Setting up odbcinst.ini

      The settings in this file impact all of the ODBC drivers on the machine.

Use an ASCII editor to update this file. It must have a stanza (section) called [IBM DB2 ODBC DRIVER], with a line starting with ″Driver″ indicating the full path to the DB2 ODBC driver (db2.o). For example, if the home directory of your end user is /u/thisuser/ and the sqllib directory is installed there, then the correct entry would be:

```
[IBM DB2 ODBC DRIVER]
Driver=/u/thisuser/sqllib/lib/db2.o
```

Setting up odbc.ini

The settings in this file are associated with a particular user on the machine; different users can have different odbc.ini files.

The .odbc.ini file must exist in the end user's home directory (note the dot at the start of the file name). Update this file, using an ASCII editor, to reflect the appropriate data source configuration information. To register a DB2 database as an ODBC data source there must be one stanza (section) for each DB2 database.

The .odbc.ini file must contain the following lines:

- in the [ODBC Data Source] stanza:

  ```
  SAMPLE=IBM DB2 ODBC DRIVER
  ```

  Indicates that there is a data source called SAMPLE that used the IBM DB2 ODBC DRIVER.

- in the [SAMPLE] stanza:

  ```
  [SAMPLE]
  Driver=/u/thisuser/sqllib/lib/db2.o
  Description=Sample DB2 ODBC Database
  ```

  Indicates that the SAMPLE database is part of the DB2 instance located in the directory /u/thisuser.

- in the [ODBC] stanza:

  ```
  InstallDir=/u/thisuser/sqllib/odbclib
  ```

  Indicates that /u/thisuser/sqllib/odbclib should be treated as the location where ODBC is installed.

- Ensure that the InstallDir correctly points to the ODBC Driver Manager location.

  For example, if the ODBC Driver Manager has been installed in /opt/odbc, the [ODBC] stanza would look like:

  ```
  [ODBC]
  Trace=0
  TraceFile=odbctrace.out
  InstallDir=/opt/odbc
  ```

See the sample file in the sqllib/odbclib subdirectory for an example. You can also see "How to Configure ODBC.INI" on page 161 for more detailed information.

Once the .ini files are set up you can run your ODBC application and access DB2 databases. Refer to the documentation that comes with your ODBC application for additional help and information.

4. Configure the DB2 CLI/ODBC driver (Optional).

   There are various keywords and values that can be used to modify the behavior of DB2 CLI/ODBC and the applications using it. The keywords are associated with the database *alias name*, and affect all DB2 CLI/ODBC applications that access the database.

   For information on manually editing this file (db2cli.ini), see "Configuring db2cli.ini" on page 159. For information about the specific keywords see "Configuration Keywords" on page 164.

Detailed Configuration Information

The section "Platform Specific Details for CLI/ODBC Access" on page 151 should provide you with all of the information you require. The following additional information is useful where DB2 tool support is not available, and for administrators who require more detailed information.

- "How to Bind the DB2 CLI/ODBC Driver to the Database"
- "How to Set CLI/ODBC Configuration Keywords" on page 158
- "Configuring db2cli.ini" on page 159

How to Bind the DB2 CLI/ODBC Driver to the Database

The CLI/ODBC driver will autobind on the first connection to the database, provided the user has the appropriate privilege or authorization. The administrator may want to perform the first connect or explicitly bind the required files.

Table 10. DB2 CLI Bind Files and Package Names

Bind File Name	Package Name	Needed by DB2 Universal Database	Needed by DRDA servers
db2clish.bnd	SQLLFyxx	Yes	Yes
db2clisn.bnd	SQLLCyxx	Yes	Yes
db2clibh.bnd	SQLLDyxx	Yes	Yes
db2clihn.bnd	SQLLEyxx	Yes	Yes
db2cliws.bnd	SQLL65zz	Version 2 or later	No

Table 10. DB2 CLI Bind Files and Package Names (continued)

Bind File Name	Package Name	Needed by DB2 Universal Database	Needed by DRDA servers
db2clims.bnd	SQLL75zz	No	DB2 for MVS/ESA
db2clivm.bnd	SQLL85zz	No	SQL/DS
db2cliv1.bnd	SQLLB5zz	Version 1 only	No
db2cliv2.bnd	SQLL95zz	Version 2 or later	No
db2clias.bnd	SQLLA5zz	No	DB2 Universal Database for AS/400

Note:

- Where 'xx' is a hexidecimal value between 00 - FF.
- Where 'y' ranges between 0 - 4.
- Where 'zz' is unique for each platform.

Previous versions of DB2 servers do not need all of the bind files and will therefore return errors at bind time.

The db2cli.lst file contains the names of the required bind files for DB2 CLI to connect to DB2 Version 2 or later servers (db2clixx.bnd where xx is `cs`, `rr`, `rs`, `ur`, `ws`, and `v2`). The db2cli1.lst file contains the names of the required bind files for DB2 CLI to connect to DB2 Version 1 servers (db2clixx.bnd where xx is `cs`, `rr`, `ur`, and `v1`).

For DRDA servers:

- use one of ddcsvm.lst, ddcsmvs.lst, ddcsvse.lst, or ddcs400.lst bind list files.
- Refer to the SYSSCHEMA keyword in "Configuration Keywords" on page 164.
- Refer to the *Quick Beginnings* or *DB2 Connect Enterprise Edition for OS/2 and Windows NT Quick Beginnings* for details about required bind options.

How to Set CLI/ODBC Configuration Keywords

DB2 CLI can be configured further by using either the CCA or the DB2 Client Setup administration tool, whichever is applicable for your platform, or by manually editing the db2cli.ini file.

This file contains various keywords and values that can be used to modify the behavior of DB2 CLI and the applications using it. The keywords are associated with the database *alias name*, and affect all DB2 CLI and ODBC applications that access the database.

By default, the location of the CLI/ODBC configuration keyword file is located in the sqllib directory on Intel platforms, and in the sqllib/cfg directory of the database instance running the CLI/ODBC applications on UNIX platforms.

The environment variable *DB2CLIINIPATH* can also be used to override the default and specify a different location for the file.

The configuration keywords enable you to:

- Configure general features such as data source name, user name, and password.
- Set options that will affect performance.
- Indicate query parameters such as wild card characters.
- Set patches or work-arounds for various ODBC applications.
- Set other, more specific features associated with the connection, such as code pages and IBM Graphic data types.

For a complete description of all the keywords and their usage, refer to "Configuration Keywords" on page 164.

Configuring db2cli.ini: The db2cli.ini initialization file is an ASCII file which stores values for the DB2 CLI configuration options. A sample file is shipped to help you get started. Refer to "Configuration Keywords" on page 164 for information on each keyword.

See "Platform Specific Details for CLI/ODBC Access" on page 151 for more information on how to modify this file on your platform.

There is one section within the file for each database (data source) the user wishes to configure. If needed, there is also a common section that affects all connections to DB2.

Only the keywords that apply to all connections to DB2 through the DB2 CLI/ODBC driver are included in the COMMON section. This includes the following keywords:

- DISABLEMULTITHREAD
- TRACE
- TRACEFILENAME
- TRACEFLUSH
- TRACEPATHNAME

All other keywords are to be placed in the database specific section, described below.

The COMMON section of the db2cli.ini file begins with:

```
[COMMON]
```

Before setting a common keyword it is important to evaluate its impact on all DB2 CLI/ODBC connections from that client. A keyword such as TRACE, for instance, will generate information on all DB2 CLI/ODBC applications connecting to DB2 on that client, even if you are intending to troubleshoot only one of those applications.

Each database specific section always begins with the name of the database alias between square brackets:

```
[
database alias]
```

This is called the ***section header***.

The parameters are set by specifying a keyword with its associated keyword value in the form:

```
KeywordName =keywordValue
```

- All the keywords and their associated values for each database must be located below the database section header.
- The keyword settings in each section apply only to the database alias named in that section header.
- The keywords are not case sensitive; however, their values can be if the values are character based.
- For the syntax associated with each keyword, refer to "DB2 CLI/ODBC Configuration Keyword Listing" on page 164.
- If a database is not found in the .INI file, the default values for these keywords are in effect.
- Comment lines are introduced by having a semi-colon in the first position of a new line.
- Blank lines are permitted.
- If duplicate entries for a keyword exist, the first entry is used (and no warning is given).

The following is a sample .INI file with 2 database alias sections:

```
; This is a comment line.
[MYDB22]
AUTOCOMMIT=0
TABLETYPE="'TABLE','SYSTEM TABLE'"

; This is another comment line.
[MYDB2MVS]
```

```
DBNAME=SAAID
TABLETYPE="'TABLE'"
SCHEMALIST="'USER1',CURRENT SQLID,'USER2'"
```

Although you can edit the db2cli.ini file manually on all platforms, we recommend that you use the CCA if it is available on your platform.

How to Configure ODBC.INI

Microsoft's 16-bit ODBC Driver Manager and all non-Microsoft ODBC Driver Managers use the odbc.ini file to record information about the available drivers and data sources. ODBC Driver Managers on UNIX platforms also uses the odbcinst.ini file. Although the necessary files are updated automatically by the tools on most platforms, users of ODBC on UNIX platforms will have to edit them manually. The file odbc.ini (and odbcinst.ini where required) are located:

UNIX Home directory of the user ID running the ODBC application (on UNIX the odbc.ini file name has a dot before it: .odbc.ini)

It is also possible to modify this file manually. Do not change any of the existing entries in the file. To edit this file manually perform the following steps:

Step 1. Use an ASCII editor to edit the odbc.ini file.

The following is an example odbc.ini file:

```
[ODBC Data Sources]
MS Access Databases=Access Data (*.mdb)

[MS Access Databases]
Driver=D:\WINDOWS\SYSTEM\simba.dll
FileType=RedISAM
SingleUser=False
UseSystemDB=False
```

The [ODBC Data Sources] section lists the name of each available data source and the description of the associated driver.

For each data source listed in the [ODBC Data Sources] section, there is a section that lists additional information about that data source. These are called the *Data Source Specification* sections.

Step 2. Under the [ODBC DATA SOURCE] entry, add the following line:

database_alias=IBM DB2 ODBC DRIVER

where *database_alias* is the alias of the database cataloged in the database directory (the database name used by the command line processor CONNECT TO statement).

Step 3. Add a new entry in the Data Source Specification section to associate the data source with the driver:

```
[database_alias]
Driver=x:\windows\system\db2cliw.dll
```

where:

- *database_alias* is the alias of the database cataloged in the database directory, and listed under the Data Source Specification section.
- *x:* is the drive where the Windows operating system is installed.

The following shows the example file with the IBM data source entries added:

```
[ODBC Data Sources]
MS Access Databases=Access Data (*.mdb)
SAMPLE=IBM DB2 ODBC DRIVER

[MS Access Databases]
Driver=D:\WINDOWS\SYSTEM\simba.dll
FileType=RedISAM
SingleUser=False
UseSystemDB=False

[SAMPLE]
Driver=D:\WINDOWS\SYSTEM\db2cliw.dll
Description=Sample DB2 Client/Server database
```

UNIX Configuration of .ini files

The section "UNIX Client Access to DB2 using CLI/ODBC" on page 155 contains detailed steps on how to update both the odbc.ini and odbcinst.ini files.

Application Development Environments

DB2 CLI application development support is provided when you install a DB2 SDK. The DB2 SDK requires the same initial runtime setup as any other DB2 client. Before using the DB2 CLI development environment, you may want to verify that your environment is set up correctly by following these steps:

- Start the command line processor by issuing the command `DB2`, or use the Command Center if it is available on your platform.
- List the cataloged databases with the command: `LIST DATABASE DIRECTORY`
- Connect to the database *database_name* with the command: `CONNECT TO` *database_name* `USER` *userid* `USING` *password*

If there are no databases cataloged, or the connect fails refer to the *Quick Beginnings* book for your platform for information about configuring the environment.

Once a connection has been confirmed, proceed with compiling the sample application.

Note: There is also an applet referred to as ″Interactive CLI″ or `db2cli` in the DB2 CLI samples directory. Refer to the `INTCLI.DOC` file in the same directory for more information on this programmer's utility that is used to design and prototype CLI function calls.

Compiling a Sample Application

DB2 CLI includes various sample applications located in the following location:

- On OS/2 and Windows 32-bit operating systems: `%DB2PATH%\samples\cli` (where `%DB2PATH%` is a variable that determines where DB2 is installed)
- On UNIX: `$HOME/sqllib/samples` (where $HOME is the home directory of the instance owner)

The `README` file in the same directory lists each sample along with an explanation, and describes how to build the samples using the makefile and build files provided. All of the DB2 CLI samples are listed in "Appendix J. CLI Sample Code" on page 847.

The *Application Building Guide* provides information for compiling, linking, and running DB2 CLI applications and stored procedures on all DB2-supported operating systems. See the section DB2 Call Level Interface (CLI) Applications in that book for complete details and examples.

If you are migrating applications from previous versions of DB2 CLI, refer to "Incompatibilities" on page 768 for additional information.

Note: The order in which the compiler searches for include (header) files can be significant if there are two or more files with the same name, (for example `sql.h` and `sqlext.h` are included in some ODBC SDK environments).

If you are building only DB2 CLI applications, always put the DB2 include path before any others.

If you are building ODBC applications, your build environment may require further customization in order to use the correct include files. Refer to the ODBC SDK documentation for more information.

Compile and Link Options

Refer to the *Application Building Guide* for information on compiling and linking the samples for your particular platform. A make file and build script are also supplied that have the correct options for the platform and supported compilers. These files are located in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory.

Note: There may be platform or compiler specific link options required for some types of CLI applications. Refer to the *Application Building Guide* for complete details.

DB2 CLI/ODBC Configuration Keyword Listing

For specific details on how to set the DB2 CLI/ODBC configuration keywords for your platform see the last step of "Platform Specific Details for CLI/ODBC Access" on page 151. See "How to Set CLI/ODBC Configuration Keywords" on page 158 for information on the location and format of the db2cli.ini file.

Configuration Keywords

The keywords are listed in alphabetical order starting with "APPENDAPINAME" on page 167. They are also divided into categories. Each of these categories is presented on a separate tab on the CLI/ODBC Settings notebook, accessible from the Client Configuration Assistant (not available on UNIX platforms).

Configuration Keywords by Category

CLI/ODBC Settings General Configuration Keywords: General keywords.

- "DBALIAS" on page 179
- "PWD" on page 196
- "UID" on page 206

Compatibility Configuration Keywords: The **Compatibility** set of options are used to define DB2 behavior. They can be set to ensure that other applications are compatible with DB2.

- "DEFERREDPREPARE" on page 182
- "DISABLEMULTITHREAD" on page 183
- "EARLYCLOSE" on page 183

Data Type Configuration Keywords: The **Data Type** set of options are used to define how DB2 reports and handles various data types.

Enterprise Configuration Keywords: The **Enterprise** set of options are used to maximize the efficiency of connections to large databases.

Environment Configuration Keywords: The **Environment** set of options are used to define the location of various files on the server and client machines.

Optimization Configuration Keywords: The **Optimization** set of options are used to speed up and reduce the amount of network flow between the CLI/ODBC Driver and the server.

- "DB2OPTIMIZATION" on page 179
- "KEEPSTATEMENT" on page 188
- "OPTIMIZEFORNROWS" on page 193
- "OPTIMIZESQLCOLUMNS" on page 193
- "UNDERSCORE" on page 207

Service Configuration Keywords: The **Service** set of options are used to help in troubleshooting problems with CLI/ODBC connections. Some options can also be used by programmers to gain a better understanding of how their CLI programs are translated into calls to the server.

- "APPENDAPINAME" on page 167
- "IGNOREWARNINGS" on page 186
- "IGNOREWARNLIST" on page 187
- "PATCH1" on page 194
- "PATCH2" on page 195
- "POPUPMESSAGE" on page 195
- "SQLSTATEFILTER" on page 197
- "TRACE" on page 201
- "TRACECOMM" on page 202
- "TRACEFILENAME" on page 203
- "TRACEFLUSH" on page 204
- "TRACEPATHNAME" on page 205
- "WARNINGLIST" on page 208

Transaction Configuration Keywords: The **Transaction** set of options are used to control and speed up SQL statements used in the application.

- "ASYNCENABLE" on page 167
- "CONNECTTYPE" on page 170
- "CURSORHOLD" on page 174
- "KEEPCONNECT" on page 188
- "MAXCONN" on page 190
- "MODE" on page 191
- "MULTICONNECT" on page 192
- "SYNCPOINT" on page 198

• “TXNISOLATION” on page 205

APPENDAPINAME

Keyword Description:
Append the CLI/ODBC function name which generated an error to the error message.

db2cli.ini Keyword Syntax:
APPENDAPINAME = **0** | **1**

Default Setting:
Do NOT display DB2 CLI function name.

DB2 CLI/ODBC Settings Tab:
Service

Usage Notes:

The DB2 CLI function (API) name that generated an error is appended to the error message retrieved using `SQLGetDiagRec()` or `SQLError()`. The function name is enclosed in curly braces { }.

For example,

```
[IBM][CLI Driver]" CLIxxxx: < text >
SQLSTATE=XXXXX {SQLGetData}"
```

0 = do NOT append DB2 CLI function name (default)

1 = append the DB2 CLI function name

This keyword is only useful for debugging.

ASYNCENABLE

Keyword Description:
Enable or disable the ability to execute queries asynchronously.

db2cli.ini Keyword Syntax:
ASYNCENABLE = **1** | **0**

Default Setting:
Execute queries asynchronously.

DB2 CLI/ODBC Settings Tab:
Transaction

Equivalent Statement Attribute:
SQL_ATTR_ASYNC_ENABLE

Usage Notes:

This option allows you to enable or disable the ability to execute queries asynchronously. This only benefits applications that were written to take advantage of this feature. Disable it only if your application does not function properly when enabled. It is placed in the data source specific section of the `db2cli.ini` file.

1 = Execute queries asynchronously (default)

0 = Queries not executed asynchronously

Note: The CLI/ODBC driver will act as it did with previous versions of DB2 that did not support asynchronous ODBC.

BITDATA

Keyword Description:

Specify whether binary data types are reported as binary or character data types.

db2cli.ini Keyword Syntax:

BITDATA = **1** | **0**

Default Setting:

Report `FOR BIT DATA` and BLOB data types as binary data types.

DB2 CLI/ODBC Settings Tab:

Data Type

Usage Notes:

This option allows you to specify whether ODBC binary data types (SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY, and SQL_BLOB), are reported as binary type data. IBM DBMSs support columns with binary data types by defining CHAR, VARCHAR, and LONG VARCHAR columns with the `FOR BIT DATA` attribute. DB2 Universal Database will also support binary data via the BLOB data type (in this case it is mapped to a CLOB data type).

Users may also need to set this option if they are using a DB2 Version 1 application that retrieves (LONG) (VAR)CHAR data into SQL_C_CHAR buffer. In DB2 Version 1, data is moved into the SQL_C_CHAR buffer unchanged; starting in DB2 Version 2, the data is converted into the ASCII representation of each hexadecimal nibble.

Only set BITDATA = 0 if you are sure that all columns defined as FOR BIT DATA or BLOB contain only character data, and the application is incapable of displaying binary data columns.

1 = report FOR BIT DATA and BLOB data types as binary data types (default).

0 = report FOR BIT DATA and BLOB data types as character data types.

CLISCHEMA

Keyword Description:
Set the DB2 ODBC catalog view to use.

db2cli.ini Keyword Syntax:
CLISCHEMA = *ODBC catalog view*

Default Setting:
None - No ODBC catalog view is used

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

See Also:
"SYSSCHEMA" on page 199

Equivalent Connection Attribute:
SQL_ATTR_CLISCHEMA

Usage Notes:

The DB2 ODBC catalog is designed to improve the performance of schema calls for lists of tables in ODBC applications that connect to host DBMSs through DB2 Connect.

The DB2 ODBC catalog, created and maintained on the host DBMS, contains rows representing objects defined in the real DB2 catalog, but these rows include only the columns necessary to support ODBC operations. The tables in the DB2 ODBC catalog are pre-joined and specifically indexed to support fast catalog access for ODBC applications.

System administrators can create multiple DB2 ODBC catalog views, each containing only the rows that are needed by a particular user group. Each end user can then select the DB2 ODBC catalog view they wish to use (by setting this keyword).

Use of the CLISCHEMA setting is completely transparent to the ODBC application; you can use this option with any ODBC application.

While this keyword has some similar effects as the SYSSCHEMA keyword, CLISCHEMA should be used instead (where applicable).

CLISCHEMA improves data access efficiency: The user-defined tables used with SYSSCHEMA were mirror images of the DB2 catalog tables, and the ODBC driver still had to join rows from multiple tables to produce the information required by the ODBC user. Using CLISCHEMA also results in less contention on the catalog tables.

CONNECTNODE

Keyword Description:
Specify the node to which a connect is to be made

db2cli.ini Keyword Syntax:
CONNECTNODE = **integer value from 1 to 999** | **SQL_CONN_CATALOG_NODE**

Default Setting:
Logical node which is defined with port 0 on the machine is used.

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

Only Applicable when:
Connecting to a multi-node DB2 Extended Enterprise Edition database server.

Equivalent Connection Attribute:
SQL_ATTR_CONNECT_NODE

Usage Notes:

Used to specify the target logical node of a DB2 Extended Enterprise Edition database partition server that you want to connect to. This keyword (or attribute setting) overrides the value of the environment variable DB2NODE. Can be set to:

- an integer between 0 and 999
- SQL_CONN_CATALOG_NODE

If this variable is not set, the target logical node defaults to the logical node which is defined with port 0 on the machine.

CONNECTTYPE

Keyword Description:
Remote or Distributed unit of work.

db2cli.ini Keyword Syntax:
CONNECTTYPE = **1** | **2**

Default Setting:
Remote unit of work

DB2 CLI/ODBC Settings Tab:
Transaction

See Also:
"SYNCPOINT" on page 198

Equivalent Connection Attribute:
SQL_ATTR_CONNECTTYPE

Usage Notes:

This option allows you to specify the default connect type.

1 = Remote unit of work. Multiple concurrent connections, each with its own commit scope. The concurrent transactions are not coordinated. (default)

2 = Distributed unit of work. Coordinated connections where multiple databases participate under the same distributed unit of work. This setting works in conjunction with the SYNCPOINT setting to determine if a Transaction Manager should be used.

CURRENTFUNCTIONPATH

Keyword Description:
Specify the schema used to resolve function references and data type references in dynamic SQL statements.

db2cli.ini Keyword Syntax:
CURRENTFUNCTIONPATH = *current_function_path*

Default Setting:
See description below.

DB2 CLI/ODBC Settings Tab:
Environment

Usage Notes:

This keyword defines the path used to resolve function references and data type references that are used in dynamic SQL statements. It contains a list of one or more schema-names, where schema-names are enclosed in double quotes and separated by commas.

The default value is ″SYSIBM″,″SYSFUN″,X where X is the value of the USER special register delimited by double quotes. The schema SYSIBM does not need to be specified. If it is not included in the function path, then it is implicitly assumed as the first schema.

This keyword is used as part of the process for resolving unqualified function references that may have been defined in a schema name other than the current user's schema. The order of the schema names determines the order in which the function names will be resolved. For more information on function resolution, refer to the *SQL Reference*.

CURRENTPACKAGESET

Keyword Description:
Issue ″SET CURRENT PACKAGESET schema″ after every connect.

db2cli.ini Keyword Syntax:
CURRENTPACKAGESET = *schema name*

Default Setting:
The clause is not appended.

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

Only Applicable when:
connecting to a DB2 for MVS/ESA v4.1 or later database.

Equivalent Connection Attribute:
SQL_ATTR_CURRENT_PACKAGE_SET

Usage Notes:

This option will issue the command ″SET CURRENT PACKAGESET schema″ after every connect to a DB2 for MVS/ESA v4.1 or later database. By default this clause is not appended.

This statement sets the schema name (collection identifier) that will be used to select the package to use for subsequent SQL statements.

CLI/ODBC applications issue dynamic SQL statements. Using this option you can control the privileges used to run these statements:

- Choose a schema to use when running SQL statements from CLI/ODBC applications.
- Ensure the objects in the schema have the desired privileges and then rebind accordingly.

- Set the CURRENTPACKAGESET option to this schema.

The SQL statements from the CLI/ODBC applications will now run under the specified schema and use the privileges defined there.

Refer to the *SQL Reference* for more information on the SET CURRENT PACKAGESET command.

CURRENTREFRESHAGE

Keyword Description:
Set the value of the CURRENT REFRESH AGE special register.

db2cli.ini Keyword Syntax:
CURRENTREFRESHAGE = **0** | **ANY** | **a numeric constant**

Default Setting:
0 - summary tables defined with REFRESH DEFERRED will not be used to optimize the processing of a query

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

Usage Notes:

For information on Summary Tables and the SET CURRENT REFRESH AGE statement, see the SQL Reference.

This keyword can be set to one of the following values:

- 0 - Indicates that summary tables defined with REFRESH DEFERRED will not be used to optimize the processing of a query (default).
- 99999999999999 - Indicates that any summary tables defined with REFRESH DEFERRED or REFRESH IMMEDIATE may be used to optimize the processing of a query. This value represents 9999 years, 99 months, 99 days, 99 hours, 99 minutes, and 99 seconds.
- ANY - This is a shorthand for 99999999999999.

CURRENTSCHEMA

Keyword Description:
Specify the schema used in a SET CURRENT SCHEMA statement upon a successful connect.

db2cli.ini Keyword Syntax:
CURRENTSCHEMA = *schema name*

Default Setting:
No statement is issued.

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

Usage Notes:

Upon a successful connect, if this option is set, a SET CURRENT SCHEMA statement is sent to the DBMS. This allows the end user or application to name SQL objects without having to qualify them by schema name.

For more information on the SET CURRENT SCHEMA statement, see the *SQL Reference*.

CURRENTSQLID

Keyword Description:
Specify the ID used in a SET CURRENT SQLID statement sent to the DBMS upon a successful connect.

db2cli.ini Keyword Syntax:
CURRENTSQLID = *current_sqlid*

Default Setting:
No statement is issued.

DB2 CLI/ODBC Settings Tab:
Enterprise

Only Applicable when:
connecting to those DB2 DBMS's where SET CURRENT SQLID is supported (such as DB2 for MVS/ESA).

Usage Notes:

Upon a successful connect, if this option is set, a SET CURRENT SQLID statement is sent to the DBMS. This allows the end user and the application to name SQL objects without having to qualify them by schema name.

CURSORHOLD

Keyword Description:
Effect of a transaction completion on open cursors.

db2cli.ini Keyword Syntax:
CURSORHOLD = **1** | **0**

Default Setting:
Selected--Cursors are not destroyed.

DB2 CLI/ODBC Settings Tab:
Transaction

Equivalent Statement Attribute:
SQL_ATTR_CURSOR_HOLD

Usage Notes:

This option controls the effect of a transaction completion on open cursors.

1 = cursor hold, the cursors are not destroyed when the transaction is committed (default).

0 = cursor no hold, the cursors are destroyed when the transaction is committed.

Note: Cursors are always destroyed when transactions are rolled back.

This option affects the result returned by `SQLGetInfo()` when called with SQL_CURSOR_COMMIT_BEHAVIOR or SQL_CURSOR_ROLLBACK_BEHAVIOR. The value of CURSORHOLD is ignored if connecting to DB2 for VSE & VM where cursor `with hold` is not supported.

You can use this option to tune performance. It can be set to cursor no hold (0) if you are sure that your application:

1. Does not have behavior that is dependent on the SQL_CURSOR_COMMIT_BEHAVIOR or the SQL_CURSOR_ROLLBACK_BEHAVIOR information returned via `SQLGetInfo()`, and
2. Does not require cursors to be preserved from one transaction to the next.

The DBMS will operate more efficiently, as resources no longer need to be maintained after the end of a transaction.

DB2CONNECTVERSION

Keyword Description:
Specify DB2 Connect or DB2 DDCS gateway version being used.

db2cli.ini Keyword Syntax:
DB2CONNECTVERSION = *gateway version*

Default Setting:
5

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

Only Applicable when:
connecting to a data source through a DB2 Connect or DB2 DDCS gateway.

Usage Notes:

This option is used to indicate to the DB2 CLI driver which version of a DB2 Connect or DB2 DDCS gateway is being used. The CLI driver can then use this information to maximize its interaction with the data source (supporting stored procedures that return multiple result sets, for instance).

5 = Indicates that a version 5 DB2 Connect gateway is being used (default).

2 = Indicates that a version 2 DB2 DDCS gateway is being used.

DB2DEGREE

Keyword Description:
Set the degree of parallelism for the execution of SQL statements.

db2cli.ini Keyword Syntax:
DB2DEGREE = **0** | **integer value from 1 to 32767** | **ANY**

Default Setting:
No SET CURRENT DEGREE statement is issued.

DB2 CLI/ODBC Settings Tab:
Optimization

Only Applicable when:
connecting to a cluster database system.

Usage Notes:

This option only applies to a DB2 Version 5.2 or later server. If the value specified is anything other than 0 (the default) then DB2 CLI will issue the following SQL statement after a successful connection:

```
SET CURRENT DEGREE value
```

This specifies the degree of parallelism for the execution of the SQL statements. The database manager will determine the degree of parallelism if you specify ANY.

For more information, see the SET CURRENT DEGREE statement in the *SQL Reference*.

DB2ESTIMATE

Keyword Description:
Threshold for displaying CLI optimizer estimates after SQL query statement preparation.

db2cli.ini Keyword Syntax:
DB2ESTIMATE = **0** | **large positive number**

Default Setting:
Estimates are not returned.

DB2 CLI/ODBC Settings Tab:
Optimization

Only Applicable when:
a GUI application accesses a DB2 Version 2 or later server.

Equivalent Connection Attribute:
SQL_ATTR_DB2ESTIMATE

Usage Notes:

This option determines whether DB2 CLI will display a dialog box to report estimates returned by the DB2 optimizer at the end of SQL query statement preparation.

0 = Estimates are not returned (default).

large positive number = The threshold above which DB2 CLI will display the window to report estimates. This value is compared against the SQLERRD(4) field in the SQLCA associated with the PREPARE. If the value in SQLERRD(4) is greater than DB2ESTIMATE, the estimates window will appear.

The graphic window will display the optimizer estimates, along with push buttons to allow users to choose whether they wish to continue with subsequent execution of this query or cancel it.

The recommended value for DB2ESTIMATE is 60000.

This option is only relevant when connecting to a DB2 version 2 or later database. In order for the window to appear, the application must have a graphical interface.

If this option is used then the DB2 CLI/ODBC option DEFERREDPREPARE will be considered off.

DB2EXPLAIN

Keyword Description:

Determines whether Explain snapshot and/or Explain table information will be generated by the server.

db2cli.ini Keyword Syntax:

DB2EXPLAIN = **0** | **1** | **2** | **3**

Default Setting:

Neither Explain snapshot nor Explain table information will be generated by the server.

DB2 CLI/ODBC Settings Tab:

Optimization

Equivalent Connection Attribute:

SQL_ATTR_DB2EXPLAIN

Usage Notes:

This keyword determines whether Explain snapshot and/or Explain table information will be generated by the server.

0 = both off (default)

A 'SET CURRENT EXPLAIN SNAPSHOT=NO' and a 'SET CURRENT EXPLAIN MODE=NO' statement will be sent to the server to disable both the Explain snapshot and the Explain table information capture facilities.

1 = Only Explain snapshot facility on

A 'SET CURRENT EXPLAIN SNAPSHOT=YES' and a 'SET CURRENT EXPLAIN MODE=NO' statement will be sent to the server to enable the Explain snapshot facility, and disable the Explain table information capture facility.

2 = Only Explain table information capture facility on

A 'SET CURRENT EXPLAIN MODE=YES' and a 'SET CURRENT EXPLAIN SNAPSHOT=NO' will be sent to the server to enable the Explain table information capture facility and disable the Explain snapshot facility.

3 = Both on

A 'SET CURRENT EXPLAIN MODE=YES' and a 'SET CURRENT EXPLAIN SNAPSHOT=YES' will be sent to the server to enable both the Explain snapshot and the Explain table information capture facilities.

Explain information is inserted into Explain tables, which must be created before the Explain information can be generated. For more information on these tables, refer to the *SQL Reference*.

The current authorization ID must have INSERT privilege for the Explain tables.

Option 1 is only valid when connecting to a DB2 Common Server version 2.1.0 or later database; options 2 and 3 when connecting to a DB2 Common Server version 2.1.1 or later database.

DB2OPTIMIZATION

Keyword Description:
Set the query optimization level.

db2cli.ini Keyword Syntax:
DB2OPTIMIZATION = *integer value from 0 to 9*

Default Setting:
No SET CURRENT QUERY OPTIMIZATION statement issued.

DB2 CLI/ODBC Settings Tab:
Optimization

Only Applicable when:
when connecting to a DB2 Version 2 server or later.

Usage Notes:

If this option is set then DB2 CLI will issue the following SQL statement after a successful connection:

```
SET CURRENT QUERY OPTIMIZATION positive number
```

This specifies the query optimization level at which the optimizer should operate the SQL queries. Refer to the *SQL Reference* for the allowable optimization levels.

DBALIAS

Keyword Description:
Enables Data Source Names greater than 8 characters.

db2cli.ini Keyword Syntax:
DBALIAS = *dbalias*

Default Setting:
Use the DB2 database alias as the ODBC Data Source Name.

DB2 CLI/ODBC Settings Tab:
CLI/ODBC Settings General

Usage Notes:

This keyword allows for Data Source Names of greater than 8 single byte characters. The Data Source Name (DSN) is the name, enclosed in square brackets, that denotes the section header in the db2cli.ini file (on platforms where this is an ASCII file). Typically, this section header is the database alias name which has a maximum length of 8 bytes. A user who wishes to refer to the data source with a longer, more meaningful name, can place the longer name in the section header, and set this keyword value to the database alias used on the CATALOG command. Here is an example:

```
; The much longer name maps to an 8 single byte character dbalias
[MyMeaningfulName]
DBALIAS=DB2DBT10
```

The end user can specify [MyMeaningfulName] as the name of the data source on connect while the actual database alias is DB2DBT10.

In a 16-bit Windows ODBC environment, under the [ODBC DATA SOURCES] entry in the ODBC.INI file, the following line must also be updated with the long alias name (*dbname*).

```
< alias >=IBM DB2 ODBC DRIVER
```

DBNAME

Keyword Description:
Specify the database name to reduce the time it takes for the application to query MVS table information.

db2cli.ini Keyword Syntax:
DBNAME = *dbname*

Default Setting:
Don't filter on the DBNAME column.

DB2 CLI/ODBC Settings Tab:
Enterprise

Only Applicable when:
connecting to DB2 for MVS/ESA.

See Also:
"SCHEMALIST" on page 196, "TABLETYPE" on page 200

Usage Notes:

This option is only used when connecting to DB2 for MVS/ESA, and only if (*base*) table catalog information is requested by the application. If a large number of tables exist in the DB2 for MVS/ESA subsystem, a *dbname* can be

specified to reduce the time it takes for the application to query table information, and reduce the number of tables listed by the application.

If this option is set then the statement IN DATABASE *dbname* will be appended to various statements such as CREATE TABLE.

This value maps to the DBNAME column in the DB2 for MVS/ESA system catalog tables. If no value is specified, or if views, synonyms, system tables, or aliases are also specified via TABLETYPE, only table information will be restricted; views, aliases, and synonyms are not restricted with DBNAME. It can be used in conjunction with SCHEMALIST, and TABLETYPE to further limit the number of tables for which information will be returned.

DEFAULTPROCLIBRARY

Keyword Description:
Set default stored procedure library.

db2cli.ini Keyword Syntax:
DEFAULTPROCLIBRARY = < *full path name* >

Default Setting:
Do not add a default stored procedure library to stored procedure calls.

DB2 CLI/ODBC Settings Tab:
Environment

Only Applicable when:
application is not using the stored procedure catalog table.

Usage Notes:

This option should only be used on a temporary basis; the stored procedure catalog table should be used instead. See the *SQL Reference* for more information.

The library pointed to by this option will be used in all stored procedure calls that do not already explicitly specify a library. Because you are specifying a location on the server machine, you must use the path format of that operating system, not of the client. For more information, see the CALL statement in the *SQL Reference*.

For instance, if the stored procedures are located on the server in the library file d:\terry\proclib\comstor, you could set DEFAULTPROCLIBRARY to d:\terry\proclib\comstor, then call the stored procedure *func* without specifying a library. The resulting SQL statement sent would be:

```
CALL d:\terry\proclib\comstor!func
```

DEFERREDPREPARE

Keyword Description:
Minimize network flow by combining the PREPARE request with the corresponding execute request.

db2cli.ini Keyword Syntax:
DEFERREDPREPARE = **0** | **1**

Default Setting:
The prepare request will be delayed until the execute request is sent.

DB2 CLI/ODBC Settings Tab:
Compatibility

Not Applicable when:
DB2ESTIMATE is set.

Equivalent Statement Attribute:
SQL_ATTR_DEFERRED_PREPARE

Usage Notes:

Defers sending the PREPARE request until the corresponding execute request is issued. The two requests are then combined into one command/reply flow (instead of two) to minimize network flow and to improve performance.

The default behavior has changed from DB2 version 2. Deferred prepare is now the default and must be explicitly turned off if required.

- 0 = Disable deferred prepare. The PREPARE request will be executed the moment it is issued.
- 1 (default) = Enable deferred prepare. Defer the execution of the PREPARE request until the corresponding execute request is issued.

  If the target DB2 Common Server database or the DDCS gateway does not support deferred prepare, the client disables deferred prepare for that connection.

Note: When deferred prepare is enabled, the row and cost estimates normally returned in the SQLERRD(3) and SQLERRD(4) of the SQLCA of a PREPARE statement may become zeros. This may be of concern to users who want to use these values to decide whether or not to continue the SQL statement.

This option is turned off if the CLI/ODBC option DB2ESTIMATE is set to a value other than zero.

DISABLEMULTITHREAD

Keyword Description:
Disable Multithreading.

db2cli.ini Keyword Syntax:
DISABLEMULTITHREAD = **0** | **1**

Default Setting:
Multithreading is enabled.

DB2 CLI/ODBC Settings Tab:
Compatibility

Usage Notes:

The CLI/ODBC driver is capable of supporting multiple concurrent threads.

This option is used to enable or disable multi-thread support.

0 = Multithreading is enabled (default).

1 = Disable Multithreading.

If multithreading is disabled then all calls for all threads will be serialized at the process level. Use this setting for multithreaded applications that require the serialized behavior of DB2 Version 2.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

EARLYCLOSE

Keyword Description:
Should the cursor associated with the connection be closed early by the DB2 server when it encounters the end of the result set?

db2cli.ini Keyword Syntax:
EARLYCLOSE = **1** | **0**

Default Setting:
EARLYCLOSE behavior is on.

DB2 CLI/ODBC Settings Tab:
Compatibility

Equivalent Statement Attribute:
SQL_ATTR_EARLYCLOSE

Usage Notes:

This option specifies whether or not the temporary cursor on the server can be automatically closed, without closing the cursor on the client, when the last record is sent to the client.

0 = Do not close the temporary cursor on the server early.

1 = Close the temporary cursor on the server early (default).

This saves the CLI/ODBC driver a network request by not issuing the statement to explicitly close the cursor because it knows that it has already been closed.

Having this option on will speed up applications that make use of many small result sets.

The EARLYCLOSE feature is not used if either:

– The statement disqualifies for blocking.
– The cursor type is anything other than SQL_CURSOR_FORWARD_ONLY.

Note: Although this option can be set at any time, the option value used is the one that exists when the statement is executed (when the cursor is opened).

GRANTEELIST

Keyword Description:

Reduce the amount of information returned when the application gets a list of table or column privileges.

db2cli.ini Keyword Syntax:

GRANTEELIST = " '*userID1*', '*userID2*',... '*userIDn*' "

Default Setting:

Do not filter the results.

DB2 CLI/ODBC Settings Tab:

Enterprise

See Also:

"GRANTORLIST" on page 185

Usage Notes:

This option can be used to reduce the amount of information returned when the application gets a list of privileges for tables in a database, or columns in a table. The list of authorization IDs specified is used as a filter; the only tables or columns that are returned are those with privileges that have been granted *TO* those IDs.

Set this option to a list of one or more authorization IDs that have been granted privileges, delimited with single quotes, and separated by commas. The entire string must also be enclosed in double quotes. For example:

```
GRANTEELIST=" 'USER1', 'USER2', 'USER8' "
```

In the above example, if the application gets a list of privileges for a specific table, only those columns that have a privilege granted *TO* USER1, USER2, or USER8 would be returned.

GRANTORLIST

Keyword Description:

Reduce the amount of information returned when the application gets a list of table or column privileges.

db2cli.ini Keyword Syntax:

GRANTORLIST = *" 'userID1', 'userID2',... 'userIDn' "*

Default Setting:

Do not filter the results.

DB2 CLI/ODBC Settings Tab:

Enterprise

See Also:

"GRANTEELIST" on page 184

Usage Notes:

This option can be used to reduce the amount of information returned when the application gets a list of privileges for tables in a database, or columns in a table. The list of authorization IDs specified is used as a filter; the only tables or columns that are returned are those with privileges that have been granted *BY* those IDs.

Set this option to a list of one or more authorization IDs that have granted privileges, delimited with single quotes, and separated by commas. The entire string must also be enclosed in double quotes. For example:

```
GRANTORLIST=" 'USER1', 'USER2', 'USER8' "
```

In the above example, if the application gets a list of privileges for a specific table, only those columns that have a privilege granted *BY* USER1, USER2, or USER8 would be returned.

GRAPHIC

Keyword Description:
Controls whether DB2 CLI reports the IBM GRAPHIC (double byte character support) as one of the supported data types.

db2cli.ini Keyword Syntax:
GRAPHIC = **0** | **1** | **2** | **3**

Default Setting:
GRAPHIC is not returned as a supported data type.

DB2 CLI/ODBC Settings Tab:
Data Type

Usage Notes:

This option controls how two related pieces of information are returned by the application:

- Whether DB2 CLI reports the IBM GRAPHIC (double byte character support) as one of the supported data types when `SQLGetTypeInfo()` is called. `SQLGetTypeInfo()` lists the data types supported by the DB2 database in the current connection.
- What unit is used to report the length of graphic columns. This applies to all DB2 CLI/ODBC functions that return length/precision either on the output argument or as part of the result set.

  0 = Do not report IBM GRAPHIC data type as a supported type. Length of graphic columns returned as number of DBCS characters. (default)

  1 = Report IBM GRAPHIC data type as supported. Length of graphic columns returned as number of DBCS characters.

  2 = Do not report IBM GRAPHIC data type as a supported type. Length of graphic columns returned as number of bytes. (This is needed for **Microsoft Access** 1.1-J** and **Microsoft Query**-J**.)

  3 = Settings 1 and 2 combined. IBM GRAPHIC data type reported as supported. Length of graphic columns returned as number of bytes.

The default is that GRAPHIC is not returned since many off the shelf applications do not recognize this data type and cannot provide proper handling.

IGNOREWARNINGS

Keyword Description:
Ignore Warnings.

db2cli.ini Keyword Syntax:
IGNOREWARNINGS = **0** | **1**

Default Setting:
Warnings are returned as normal.

DB2 CLI/ODBC Settings Tab:
Service

See Also:
"WARNINGLIST" on page 208, "IGNOREWARNLIST"

Usage Notes:

On rare occasions an application will not correctly handle warning messages. This option can be used to indicate that warnings from the database manager are not to be passed on to the application.

0 = Warnings reported as usual (default).

1 = Database manager warnings are ignored, SQL_SUCCESS is returned. Warnings from the DB2 CLI/ODBC driver are still returned; many are required for normal operation.

Although this option can be used on its own, it can also be used in conjunction with the WARNINGLIST CLI/ODBC configuration keyword.

IGNOREWARNLIST

Keyword Description:
Ignore specified sqlstates.

db2cli.ini Keyword Syntax:
IGNOREWARNLIST = "'sqlstate1', 'sqlstate2', ..."

Default Setting:
Warnings are returned as normal

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

See Also:
"WARNINGLIST" on page 208, "IGNOREWARNINGS" on page 186

Usage Notes:

On rare occasions an application may not correctly handle some warning messages, but does not want to ignore all warning messages. This keyword can be used to indicate which warnings are not to be passed on to the application. The IGNOREWARNINGS keyword should be used if all database manager warnings are to be ignored.

If an sqlstate is included in both IGNOREWARNLIST and WARNINGLIST, it will be ignored altogether.

Each sqlstate must be in uppercase, delimited with single quotes and separated by commas. The entire string must also be enclosed in double quotes. For example:

```
IGNOREWARNLIST="'01000', '01004','01504'"
```

KEEPCONNECT

Keyword Description:
Number of connections to cache.

db2cli.ini Keyword Syntax:
KEEPCONNECT = **0** | **positive integer**

Default Setting:
Do not cache connections.

DB2 CLI/ODBC Settings Tab:
Transaction

Usage Notes:

0 = Do not cache database connections (default).

Setting this option to a value greater than zero can speed up applications that constantly connect to and disconnect from the same database using the same connection information.

Instead of closing the connection each time, then re-opening it again, the CLI/ODBC driver will keep the connection open and cache the connection information. When the request to connect to the same database occurs a second time, the existing connection is used. This saves the time, resources, and network flow to close the first connection, as well as to re-open the second connection.

The value set for this option indicates the number of database connections to cache. Although the maximum is limited only by system resources, usually a value of 1 or 2 is sufficient for applications that will benefit at all from this behavior.

KEEPSTATEMENT

Keyword Description:
Number of statement handles to cache.

db2cli.ini Keyword Syntax:
KEEPSTATEMENT = **5** | **positive integer**

Default Setting:
Cache 5 statement handles.

DB2 CLI/ODBC Settings Tab:
Optimization

Usage Notes:

By default, the memory required for 5 statement handles is cached. When a statement handle is closed, the memory used for that handle is not deallocated but is instead used when the next statement handle is allocated.

The value set for this option determines how many statement handles are cached. It can be set to less than 5 to explicitly reduce the amount of memory used by the statement cache. It can be increased above 5 to improve performance for applications that open, close, and then re-open large sets of statements.

The maximum number of cached statement handles is determined by system resources.

LOBMAXCOLUMNSIZE

Keyword Description:
Override default COLUMN_SIZE for LOB data types.

db2cli.ini Keyword Syntax:
LOBMAXCOLUMNSIZE = *integer greater than zero*

Default Setting:
2 Gigabytes (1G for DBCLOB)

DB2 CLI/ODBC Settings Tab:
Data Type

Only Applicable when:
LONGDATACOMPAT option is used.

See Also:
"LONGDATACOMPAT" on page 190

Usage Notes:

This will override the 2 Gigabyte (1G for DBCLOB) value that is returned by `SQLGetTypeInfo()` for the COLUMN_SIZE column for SQL_CLOB, SQL_BLOB, and SQL_DBCLOB SQL data types. Subsequent CREATE TABLE statements that contain LOB columns will use the column size value you set here instead of the default.

LONGDATACOMPAT

Keyword Description:
Report LOBs as long data types or as large object types.

db2cli.ini Keyword Syntax:
LONGDATACOMPAT = **0** | **1**

Default Setting:
Reference LOB data types as large object types.

DB2 CLI/ODBC Settings Tab:
Data Type

See Also:
"LOBMAXCOLUMNSIZE" on page 189

Equivalent Connection Attribute:
SQL_ATTR_LONGDATA_COMPAT

Usage Notes:

This option indicates to DB2 CLI what data type the application expects when working with a database with large object (LOB) columns.

Database data type	Large Objects (0--Default)	Long Data Types (1)
CLOB	SQL_CLOB	SQL_LONGVARCHAR
BLOB	SQL_BLOB	SQL_LONGVARBINARY
DBCLOB	SQL_DBCLOB	SQL_LONGVARGRAPHIC

This option is useful when running ODBC applications that cannot handle the large object data types.

The DB2 CLI/ODBC option LOBMAXCOLUMNSIZE can be used in conjunction with this option to reduce the default size declared for the data.

MAXCONN

Keyword Description:
Maximum number of connections allowed for each application.

db2cli.ini Keyword Syntax:
MAXCONN = **0** | **positive number**

Default Setting:
As many connections as permitted by system resources.

DB2 CLI/ODBC Settings Tab:
Transaction

Equivalent Connection Attribute:
SQL_ATTR_MAXCONN

Usage Notes:

This option is used to specify the maximum number of connections allowed for each CLI/ODBC application. This can be used as a governor for the maximum number of connections an administrator may wish to restrict each application to open. A value of 0 may be used to represent *no limit*; that is, an application is allowed to open up as many connections as permitted by the system resources.

On OS/2 and WIN32 platforms (Windows NT and Windows 95), if the NetBIOS protocol is in use, this value corresponds to the number of connections (NetBIOS sessions) that will be concurrently set up by the application. The range of values for OS/2 NetBIOS is 1 to 254. Specifying 0 (the default) will result in 5 *reserved* connections. *Reserved NetBIOS sessions* cannot be used by other applications. The number of connections specified by this parameter will be applied to any adapter that the DB2 NetBIOS protocol uses to connect to the remote server (adapter number is specified in the node directory for a NetBIOS node).

MODE

Keyword Description:
Default connect mode.

db2cli.ini Keyword Syntax:
MODE = **SHARE** | **EXCLUSIVE**

Default Setting:
SHARE

DB2 CLI/ODBC Settings Tab:
Transaction

Not Applicable when:
connecting to a DRDA database.

Usage Notes:

Sets the CONNECT mode to either SHARE or EXCLUSIVE. If a mode is set by the application at connect time, this value is ignored. The default is SHARE.

Note: EXCLUSIVE is not permitted for DRDA connections. Refer to the *SQL Reference* for more information on the CONNECT statement.

MULTICONNECT

Keyword Description:

How SQLConnect() requests are mapped to physical database connections.

db2cli.ini Keyword Syntax:

MULTICONNECT = **0** | **1**

Default Setting:

Each SQLConnect() request by the application will result in a physical database connection.

DB2 CLI/ODBC Settings Tab:

Transaction

Usage Notes:

This option is used to specify how SQLConnect() requests are mapped to physical database connections.

1 = Connections are not shared, multiple connections are used (default) -- Each SQLConnect() request by the application will result in a physical database connection.

0 = Connections are mapped to one physical connection, one connection is used -- All connections for the application are mapped to one physical connection. This may be useful if:

- the ODBC application runs out of file handles because it uses so many connections.
- the application only reads data from the database
- the application uses autocommit (in some cases)
- the application opens multiple connections instead of using multiple statements on one connection. The use of multiple connections in this case may cause locking contention between connections.

If MULTICONNECT is set to 0 then multithreading must be disabled using the keyword DISABLEMULTITHREAD

Note: If MULTICONNECT is set off then all statements are executed on the same connection and therefore in the same transaction. This means that a rollback will roll back ALL statements on all connections. Be sure that the application is designed to work with MULTICONNECT off before doing so or the application may not operate correctly.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

OPTIMIZEFORNROWS

Keyword Description:
Append ″OPTIMIZE FOR n ROWS″ clause to every select statement.

db2cli.ini Keyword Syntax:
OPTIMIZEFORNROWS = *integer*

Default Setting:
The clause is not appended.

DB2 CLI/ODBC Settings Tab:
Optimization

Equivalent Statement Attribute:
SQL_ATTR_OPTIMIZE_FOR_NROWS

Usage Notes:

This option will append the ″OPTIMIZE FOR n ROWS″ clause to every select statement, where n is an integer larger than 0. If set to 0 (the default) this clause will not be appended.

For more information on the effect of the OPTIMIZE FOR n ROWS clause, refer to the *Administration Guide*.

OPTIMIZESQLCOLUMNS

Keyword Description:
Optimize `SQLColumns()` call with explicit Schema and Table Name.

db2cli.ini Keyword Syntax:
OPTIMIZESQLCOLUMNS = **0** | **1**

Default Setting:
0 - all column information returned

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

Equivalent Connection Attribute:
SQL_ATTR_OPTIMIZESQLCOLUMNS

Usage Notes:

If OPTIMIZESQLCOLUMNS is on (set to 1), then all calls to `SQLColumns()` will be optimized if an explicit (no wildcard specified) Schema Name, explicit Table Name, and % (ALL columns) for Column Name are specified. The DB2

CLI/ODBC Driver will optimize this call so that the system tables will not be scanned. If the call is optimized then the COLUMN_DEF information (which contains the default string for the columns) is not returned. When connecting to an AS/400 database, the information returned by `SQLColumns()` for columns whose data type is NUMERIC will be incorrect. If the application does not need this information then it can turn on the optimization to increase performance.

If the application needs the COLUMN_DEF information then OPTIMIZESQLCOLUMNS should be set to 0. This is the default.

PATCH1

Keyword Description:
Use work-arounds for known problems with ODBC applications.

db2cli.ini Keyword Syntax:
PATCH1 = **{ 0 | 1 | 2 | 4 | 8 | 16 | ... }**

Default Setting:
Use no work-arounds.

DB2 CLI/ODBC Settings Tab:
Service

See Also:
"PATCH2" on page 195

Usage Notes:

This keyword is used to specify a work-around for known problems with ODBC applications. The value specified can be for none, one, or multiple work-arounds. The patch values specified here are used in conjunction with any PATCH2 values that may also be set.

Using the DB2 CLI/ODBC Settings notebook you can select one or more patches to use. If you set the values in the `db2cli.ini` file itself and want to use multiple patch values then simply add the values together to form the keyword value. For example, if you want the patches 1, 4, and 8, then specify PATCH1=13.

0 = No work around (default)

The DB2 CLI/ODBC Settings notebook has a list of values. Select the Service folder in the DB2 folder for information on how to update this list of values. This information is also contained in the `README` file (there will be no such section in the `README` if there are no current patch values for that platform).

PATCH2

Keyword Description:
Use work-arounds for known problems with CLI/ODBC applications.

db2cli.ini Keyword Syntax:
PATCH2 = *"patch value 1, patch value 2, patch value 3, ..."*

Default Setting:
Use no work-arounds

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

See Also:
"PATCH1" on page 194

Usage Notes:

This keyword is used to specify a work-around for known problems with CLI/ODBC applications. The value specified can be for none, one, or multiple work-arounds. The patch values specified here are used in conjunction with any PATCH1 values that may also be set.

When specifying multiple patches, the values are specified in a comma delimited string (unlike the PATCH1 option where the values are added together and the sum is used).

0 = No work around (default)

To set PATCH2 values 3, 4 and 8 you would specify:

```
PATCH2="3, 4, 8"
```

The PATCH2 values are contained in the README file (there will be no such section in the README if there are no current patch values for that platform).

POPUPMESSAGE

Keyword Description:
Pop up a message box every time CLI/ODBC generates an error.

db2cli.ini Keyword Syntax:
POPUPMESSAGE = **0** | **1**

Default Setting:
Do not display message box.

DB2 CLI/ODBC Settings Tab:
Service

Only Applicable when:
running OS/2 or Windows applications.

See Also:
"SQLSTATEFILTER" on page 197

Usage Notes:

Pops up a message box every time DB2 CLI generates an error that can be retrieved using `SQLGetDiagRec()` or `SQLError()`. Useful for debugging applications that do not report messages to users.

0 = do NOT display message box (default)

1 = display message box

PWD

Keyword Description:
Define default password.

db2cli.ini Keyword Syntax:
PWD = *password*

Default Setting:
None

DB2 CLI/ODBC Settings Tab:
CLI/ODBC Settings General

Usage Notes:

This *password* value is used if a password is not provided by the application at connect time.

It is stored as plain text and is therefore not secure.

SCHEMALIST

Keyword Description:
Restrict schemas used to query table information.

db2cli.ini Keyword Syntax:
SCHEMALIST = " *'schema1', 'schema2',... 'schemaN'* "

Default Setting:
None

DB2 CLI/ODBC Settings Tab:
Enterprise

Usage Notes:

SCHEMALIST is used to provide a more restrictive default, and therefore improve performance, for those applications that list every table in the DBMS.

If there are a large number of tables defined in the database, a schema list can be specified to reduce the time it takes for the application to query table information, and reduce the number of tables listed by the application. Each schema name is case-sensitive, must be delimited with single quotes, and separated by commas. The entire string must also be enclosed in double quotes. For example:

```
SCHEMALIST="'USER1','USER2','USER3'"
```

For DB2 for MVS/ESA, CURRENT SQLID can also be included in this list, but without the single quotes, for example:

```
SCHEMALIST="'USER1',CURRENT SQLID,'USER3'"
```

The maximum length of the string is 256 characters.

This option can be used in conjunction with DBNAME and TABLETYPE to further limit the number of tables for which information will be returned.

SQLSTATEFILTER

Keyword Description:
Do not pop up an error message for the defined SQLSTATES.

db2cli.ini Keyword Syntax:
SQLSTATEFILTER = " *'XXXXX'*, *'YYYYY'*, ... "

Default Setting:
None

DB2 CLI/ODBC Settings Tab:
Service

Only Applicable when:
POPUPMESSAGE option is turned on.

See Also:
"POPUPMESSAGE" on page 195

Usage Notes:

Use in conjunction with the POPUPMESSAGE option. This prevents DB2 CLI from displaying errors that are associated with the defined states.

Each SQLSTATE must be in upper case, delimited with single quotes and separated by commas. The entire string must also be enclosed in double quotes. For example:

```
SQLSTATEFILTER=" 'HY1090', '01504', '01508' "
```

SYNCPOINT

Keyword Description:
How commits and rollbacks are coordinated among multiple database (DUOW) connections.

db2cli.ini Keyword Syntax:
SYNCPOINT = **1** | **2**

Default Setting:
1 Phase commit.

DB2 CLI/ODBC Settings Tab:
Transaction

Only Applicable when:
default connect type set to Coordinated Connections (CONNECTTYPE=2)

See Also:
"CONNECTTYPE" on page 170

Equivalent Connection Attribute:
SQL_ATTR_SYNC_POINT

Usage Notes:

Use this option to specify how commits and rollbacks will be coordinated among multiple database (DUOW) connections. It is only relevant when the default connect type is set to Coordinated connections (CONNECTTYPE = 2).

- 1 = ONEPHASE (default)

  A Transaction Manager is not used to perform two phase commit but one phase commit is used to commit the work done by each database in a multiple database transaction.
- 2 = TWOPHASE

  A Transaction Manager is required to coordinate two phase commits among those databases that support this.

SYSSCHEMA

Keyword Description:

Indicates an alternative schema to be searched in place of the SYSIBM (or SYSTEM, QSYS2) schemas.

db2cli.ini Keyword Syntax:

SYSSCHEMA = *sysschema*

Default Setting:

No alternatives specified.

DB2 CLI/ODBC Settings Tab:

Enterprise

Usage Notes:

This option indicates an alternative schema to be searched in place of the SYSIBM (or SYSTEM, QSYS2) schemas when the DB2 CLI and ODBC Catalog Function calls are issued to obtain system catalog information.

Using this schema name the system administrator can define a set of views consisting of a subset of the rows for each of the following system catalog tables:

DB2 Universal Database	DB2 for MVS/ESA	DB2 for VSE & VM	OS/400	DB2 Universal Database for AS/400
SYSTABLES	SYSTABLES	SYSCATALOG	SYSTABLES	SYSTABLES
SYSCOLUMNS	SYSCOLUMNS	SYSCOLUMNS	SYSCOLUMNS	SYSCOLUMNS
SYSINDEXES	SYSINDEXES	SYSINDEXES	SYSINDEXES	SYSINDEXES
SYSTABAUTH	SYSTABAUTH	SYSTABAUTH		SYSCST
SYSRELS	SYSRELS	SYSKEYCOLS		SYSKEYCST
SYSDATATYPES	SYSSYNONYMS	SYSSYNONYMS		SYSCSTCOL
SYSPROCEDURES	SYSKEYS	SYSKEYS		SYSKEYS
SYSPROCPARMS	SYSCOLAUTH	SYSCOLAUTH		SYSREFCST
	SYSFOREIGNKEYS			
	SYSPROCEDURES **1**			
	SYSDATABASE			

1 DB2 for MVS/ESA 4.1 only.

For example, if the set of views for the system catalog tables is in the ACME schema, then the view for SYSIBM.SYSTABLES is ACME.SYSTABLES; and SYSSCHEMA should then be set to ACME.

Defining and using limited views of the system catalog tables reduces the number of tables listed by the application, which reduces the time it takes for the application to query table information.

If no value is specified, the default is:

- SYSCAT or SYSIBM on DB2 Universal Database
- SYSIBM on DB2 for common server versions prior to 2.1, DB2 for MVS/ESA and OS/400
- SYSTEM on DB2 for VSE & VM
- QSYS2 on DB2 Universal Database for AS/400

This keyword can be used in conjunction with SCHEMALIST and TABLETYPE (and DBNAME on DB2 for MVS/ESA) to further limit the number of tables for which information will be returned.

TABLETYPE

Keyword Description:

Define a default list of TABLETYPES returned when querying table information.

db2cli.ini Keyword Syntax:

TABLETYPE = *" 'TABLE' | 'ALIAS' | 'VIEW' | 'INOPERATIVE VIEW' | 'SYSTEM TABLE' | 'SYNONYM' "*

Default Setting:

No default list of TABLETYPES is defined.

DB2 CLI/ODBC Settings Tab:

Enterprise

Usage Notes:

If there is a large number of tables defined in the database, a tabletype string can be specified to reduce the time it takes for the application to query table information, and reduce the number of tables listed by the application.

Any number of the values can be specified. Each type must be delimited with single quotes, separated by commas, and in uppercase. The entire string must also be enclosed in double quotes. For example:

```
TABLETYPE="'TABLE','VIEW'"
```

This option can be used in conjunction with DBNAME and SCHEMALIST to further limit the number of tables for which information will be returned.

TABLETYPE is used to provide a default for the DB2 CLI function that retrieves the list of tables, views, aliases, and synonyms in the database. If the application does not specify a table type on the function call, and this keyword is not used, information about all table types is returned. If the application does supply a value for the *tabletype* on the function call, then that argument value will override this keyword value.

If TABLETYPE includes any value other than TABLE, then the DBNAME keyword setting cannot be used to restrict information to a particular DB2 for MVS/ESA database.

TEMPDIR

Keyword Description:
Define the directory used for temporary files associated with LOB fields.

db2cli.ini Keyword Syntax:
TEMPDIR = < *full path name* >

Default Setting:
Use the system temporary directory.

DB2 CLI/ODBC Settings Tab:
Environment

Usage Notes:

When working with Large Objects (CLOBS, BLOBS, etc...), a temporary file is often created on the client machine to store the information. Using this option you can specify a location for these temporary files. The system temporary directory will be used if nothing is specified.

The keyword is placed in the data source specific section of the db2cli.ini file, and has the following syntax:

- TempDir= F:\DB2TEMP

When a Large Object is accessed, an SQLSTATE of HY507 will be returned if the path name is invalid, or if the temporary files cannot be created in the directory specified.

TRACE

Keyword Description:
Turn on the DB2 CLI/ODBC trace facility.

db2cli.ini Keyword Syntax:
TRACE = **0** | **1**

Default Setting:
No trace information is captured.

DB2 CLI/ODBC Settings Tab:
Service

See Also:
"TRACEFILENAME" on page 203, "TRACEFLUSH" on page 204, "TRACEPATHNAME" on page 205

Equivalent Connection Attribute:
SQL_ATTR_TRACE

Usage Notes:

When this option is on (1), CLI/ODBC trace records are appended to the file indicated by the TRACEFILENAME configuration parameter or to files in the subdirectory indicated by the TRACEPATHNAME configuration parameter.

For example, to set up a CLI/ODBC trace file that is written to disk after each trace entry:

```
[COMMON]
TRACE=1
TRACEFILENAME=E:\TRACES\CLI\MONDAY.CLI
TRACEFLUSH=1
```

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TRACECOMM

Keyword Description:
Include information about each network request in the trace file.

db2cli.ini Keyword Syntax:
TRACECOMM = **0** | **1**

Default Setting:
0 - No network request information is captured.

DB2 CLI/ODBC Settings Tab:
This keyword cannot be set using the CLI/ODBC Settings notebook. The db2cli.ini file must be modified directly to make use of this keyword.

Only Applicable when:
the CLI/ODBC TRACE option option is turned on.

See Also:

"TRACE" on page 201, "TRACEFILENAME", "TRACEPATHNAME" on page 205, "TRACEFLUSH" on page 204

Usage Notes:

When TRACECOMM is set on (1) then information about each network request will be included in the trace file.

This option is only used when the TRACE CLI/ODBC option is turned on. See the TRACE option for an example.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TRACEFILENAME

Keyword Description:

File used to store the DB2 CLI/ODBC trace information.

db2cli.ini Keyword Syntax:

TRACEFILENAME = < **Full file name** >

Default Setting:

None

DB2 CLI/ODBC Settings Tab:

Service

Only Applicable when:

the TRACE option is turned on.

See Also:

"TRACE" on page 201, "TRACEFLUSH" on page 204, "TRACEPATHNAME" on page 205

Equivalent Connection Attribute:

SQL_ATTR_TRACEFILE

Usage Notes:

If the file specified does not exist, then it will be created; otherwise, the new trace information will be appended to the end of the file.

If the filename given is invalid or if the file cannot be created or written to, no trace will occur and no error message will be returned.

This option is only used when the TRACE option is turned on. This will be done automatically when you set this option in the CLI/ODBC Configuration utility.

See the TRACE option for an example of using the various trace settings. The TRACEPATHNAME option will be ignored if this option is set.

DB2 CLI trace should only be used for debugging purposes. It will slow down the execution of the CLI/ODBC driver, and the trace information can grow quite large if it is left on for extended periods of time.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TRACEFLUSH

Keyword Description:
Force a write to disk after each CLI/ODBC trace entry.

db2cli.ini Keyword Syntax:
TRACEFLUSH = **0** | **1**

Default Setting:
Do not write after every entry.

DB2 CLI/ODBC Settings Tab:
Service

Only Applicable when:
the CLI/ODBC TRACE option option is turned on.

See Also:
"TRACE" on page 201, "TRACEFILENAME" on page 203, "TRACEPATHNAME" on page 205

Usage Notes:

Set this option on (TRACEFLUSH = 1) to force a write to disk after each trace entry. This will slow down the trace process, but will ensure that each entry is written to disk before the application continues to the next statement.

This option is only used when the TRACE CLI/ODBC option is turned on. See the TRACE option for an example.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TRACEPATHNAME

Keyword Description:
Subdirectory used to store individual DB2 CLI/ODBC trace files.

db2cli.ini Keyword Syntax:
TRACEPATHNAME = < **Full subdirectory name** >

Default Setting:
None

DB2 CLI/ODBC Settings Tab:
Service

Only Applicable when:
the TRACE option is turned on.

Not Applicable when:
the TRACEFILENAME option is turned on.

See Also:
"TRACE" on page 201, "TRACEFILENAME" on page 203, "TRACEFLUSH" on page 204

Usage Notes:

Each thread or process that uses the same DLL or shared library will have a separate DB2 CLI/ODBC trace file created in the specified directory.

No trace will occur, and no error message will be returned, if the subdirectory given is invalid or if it cannot be written to.

This option is only used when the TRACE option is turned on. This will be done automatically when you set this option in the CLI/ODBC Configuration utility.

See the TRACE option for an example of using the various trace settings. It will be ignored if the DB2 CLI/ODBC option TRACEFILENAME is used.

DB2 CLI trace should only be used for debugging purposes. It will slow down the execution of the CLI/ODBC driver, and the trace information can grow quite large if it is left on for extended periods of time.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TXNISOLATION

Keyword Description:
Set the default isolation level.

db2cli.ini Keyword Syntax:
TXNISOLATION = **1** | **2** | **4** | **8** | **32**

Default Setting:
Read Committed (Cursor Stability)

DB2 CLI/ODBC Settings Tab:
Transaction

Only Applicable when:
the default isolation level is used. This keyword will have no effect if the application has specifically set the isolation level.

Equivalent Statement Attribute:
SQL_ATTR_TXN_ISOLATION

Usage Notes:

Sets the isolation level to:

1 = Read Uncommitted (Uncommitted read)

2 = Read Committed (Cursor stability) (default)

4 = Repeatable Read (Read Stability)

8 = Serializable (Repeatable read)

32 = (No Commit, DATABASE 2 for AS/400 only; this is similar to autocommit)

The words in parentheses are IBM's terminology for the equivalent SQL92 isolation levels. Note that *no commit* is not an SQL92 isolation level and is supported only on DB2 Universal Database for AS/400. Refer to the *SQL Reference* for more information on isolation levels.

This keyword is only applicable if the default isolation level is used. If the application has specifically set the isolation level then this keyword will have no effect.

UID

Keyword Description:
Define default user ID.

db2cli.ini Keyword Syntax:
UID = *userid*

Default Setting:
None

DB2 CLI/ODBC Settings Tab:
CLI/ODBC Settings General

Usage Notes:

The specified *userid* value is used if a userid is not provided by the application at connect time.

UNDERSCORE

Keyword Description:
Specify whether or not the underscore character "_" is to be used as a wildcard character.

db2cli.ini Keyword Syntax:
UNDERSCORE = **1** | **0**

Default Setting:
"_" acts as a wildcard.

DB2 CLI/ODBC Settings Tab:
Optimization

Usage Notes:

This option allows you to specify whether the underscore character "_" is to be used as a wildcard character (matching any one character, including no character), or to be used as itself. This option only affects catalog function calls that accept search pattern strings.

- 1 = "_" acts as a wildcard (default)

  The underscore is treated as a wildcard matching any one character or none. For example, if two tables are defined as follows:

  ```
  CREATE TABLE "OWNER"."KEY_WORDS" (COL1 INT)
  CREATE TABLE "OWNER"."KEYWORDS" (COL1 INT)
  ```

  The DB2 CLI catalog function call that returns table information (`SQLTables()`) will return both of these entries if "KEY_WORDS" is specified in the table name search pattern argument.

- 0 = "_" acts as itself

  The underscore is treated as itself. If two tables are defined as the example above, `SQLTables()` will return only the "KEY_WORDS" entry if "KEY_WORDS" is specified in the table name search pattern argument.

  Setting this keyword to 0 can result in performance improvement in those cases where object names (owner, table, column) in the database contain underscores.

Note: This keyword only has an effect on DB2 common server versions prior to Version 2.1. The ESCAPE clause for the LIKE predicate can be used

for subsequent versions and all other DB2 servers. For more information on the ESCAPE clause, refer to the *SQL Reference*.

WARNINGLIST

Keyword Description:
Specify which errors to downgrade to warnings.

db2cli.ini Keyword Syntax:
WARNINGLIST = " '**xxxxx**', '**yyyyy**', ..."

Default Setting:
Do not downgrade any SQLSTATEs.

DB2 CLI/ODBC Settings Tab:
Service

See Also:
"IGNOREWARNLIST" on page 187, "IGNOREWARNINGS" on page 186

Usage Notes:

Any number of SQLSTATEs returned as errors can be downgraded to warnings. Each must be delimited with single quotes, separated by commas, and in uppercase. The entire string must also be enclosed in double quotes. For example:

```
WARNINGLIST=" '01S02', 'HY090' "
```

This option can be used in conjunction with the IGNOREWARNINGS CLI/ODBC configuration keyword. If you also set IGNOREWARNINGS on then any errors you downgrade to warnings will not be reported at all.

Chapter 5. DB2 CLI Functions

This section provides a description of each DB2 CLI function, ordered alphabetically. There is also a table of functions by category, in the "DB2 CLI Function Summary" on page 211. Each description has the following sections.

- Status
- Purpose
- Syntax
- Arguments
- Usage
- Return Codes
- Diagnostics
- Restrictions
- Example
- References

Each section is described below.

Status of this Function since DB2 CLI Version 5

This section is only included in Version 2 functions that have been replaced with new functions in Version 5.

It describes what new function should be used, and how to use it in place of the old function.

Purpose

This section gives a brief overview of what the function does. It also indicates if any functions should be called before and after calling the function being described.

Each function also has a table, such as the one below which indicates which specification or standard the function conforms to. The first column indicates which level of DB2 CLI the function was first provided, the second column indicates which version (1.0, 2.0, or 3.0) of the ODBC specification the function was first provided. The last column indicates if the function is included in the ISO CLI standard.

Note: This table indicates support of the function, some functions use a set of options that do not apply to all specifications or standards. The restrictions section will identify any significant differences.

Table 11. Sample Function Specification Table

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

Syntax

This section contains the generic 'C' prototype. The generic prototype is used for all environments, including Windows.

Note: All function arguments that are pointers are defined using the macro FAR, this macro is defined out (set to a blank) for all platforms except Windows. On Windows FAR is used to define pointer arguments as far pointers.

Arguments

This section lists each function argument, along with its data type, a description and whether it is an input or output argument.

Only SQLGetInfo() and SQLBindParameter() have parameters that are both input and output.

Some functions contain input or output arguments which are known as *deferred* or *bound* arguments.

These arguments are pointers to buffers allocated by the application, and are associated with (or bound to) either a parameter in an SQL statement, or a column in a result set. The data areas specified by the function are accessed by DB2 CLI at a later time. It is important that these deferred data areas are still valid at the time DB2 CLI accesses them.

Usage This section provides information about how to use the function, and any special considerations. Possible error conditions are not discussed here, but are listed in the diagnostics section instead.

Return Codes

This section lists all the possible function return codes. When SQL_ERROR or SQL_SUCCESS_WITH_INFO is returned, error information can be obtained by calling SQLError().

Refer to "Diagnostics" on page 25 for more information about return codes.

Diagnostics

This section contains a table that lists the SQLSTATEs explicitly returned by DB2 CLI (SQLSTATEs generated by the DBMS may also be returned) and indicates the cause of the error. These values are obtained by calling SQLError() after the function returns a SQL_ERROR or SQL_SUCCESS_WITH_INFO.

Refer to "Diagnostics" on page 25 for more information about diagnostics.

Restrictions

This section indicates any differences or limitations between DB2 CLI and ODBC that may affect an application.

Example

This section contains either a code fragment, or a reference to a code fragment demonstrating the use of the function, using the generic data type definitions. The complete source used for all code fragments is available in the `sqllib/samples/cli` (or `sqllib\samples\cli`) directory. For a list of the included examples, refer to "Appendix J. CLI Sample Code" on page 847.

See "Chapter 4. Configuring CLI/ODBC and Running Sample Applications" on page 149 for more information on setting up the DB2 CLI environment and accessing the sample applications.

References

This section lists related DB2 CLI functions.

DB2 CLI Function Summary

Depr in the ODBC column indicates that the function has been deprecated in ODBC. See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information.

The SQL/CLI column can have the following values:

- 95 - Defined in the SQL/CLI 9075-3 specification.
- SQL3 - Defined in the SQL/CLI part of the ISO SQL3 draft replacement for SQL/CLI 9075-3.

Table 12. DB2 CLI Function List by Category

Task Function Name	ODBC 3.0	SQL/CLI	DB2 CLI First Version Supported	Purpose
Connecting to a Data Source				
SQLAllocEnv	Depr	95	V 1.1	Obtains an environment handle. One environment handle is used for one or more connections.
SQLAllocConnect	Depr	95	V 1.1	Obtains a connection handle.
`SQLAllocHandle()`	Core	95	V 5	Obtains a handle.
`SQLBrowseConnect()`	Level 1	95	V 5	Get required attributes to connect to a data source.
`SQLConnect()`	Core	95	V 1.1	Connects to specific driver by data source name, user Id, and password.

Table 12. DB2 CLI Function List by Category (continued)

Task Function Name	ODBC 3.0	SQL/CLI	DB2 CLI First Version Supported	Purpose
SQLDriverConnect()	Core	SQL3	V 2.1 [a]	Connects to a specific driver by connection string or optionally requests that the Driver Manager and driver display connection dialogs for the user. **Note:** This function is also affected by the additional IBM keywords supported in the ODBC.INI file.
SQLDrivers()	Core	No	NONE	DB2 CLI does not support this function as it is implemented by a Driver Manager.
SQLSetConnectAttr()	Core	95	V 5	Set connection attributes.
SQLSetConnectOption()	Depr	95	V 2.1	Set connection attributes.
SQLSetConnection()	No	SQL3	V 2.1	Sets the current active connection. This function only needs to be used when using embedded SQL within a DB2 CLI application with multiple concurrent connections.
DataLink Functions				
SQLBuildDataLink()	No	Yes	V 5.2	Build DATALINK Value
SQLGetDataLinkAttr()	No	Yes	V 5.2	Get DataLink attribute value
Obtaining Information about a Driver and Data Source				
SQLDataSources	Lvl 2	95	V 1.1	Returns the list of available data sources.
SQLGetInfo()	Core	95	V 1.1	Returns information about a specific driver and data source.
SQLGetFunctions()	Core	95	V 1.1	Returns supported driver functions.
SQLGetTypeInfo()	Core	95	V 1.1	Returns information about supported data types.
Setting and Retrieving Driver Options				
SQLSetEnvAttr()	Core	95	V 2.1	Sets an environment option.
SQLGetEnvAttr()	Core	95	V 2.1	Returns the value of an environment option.
SQLSetConnectOption()	Lvl 1	Yes	V 2.1 [a]	Sets a connection option.
SQLGetConnectAttr()	Lvl 1	95	V 5	Returns the value of a connection option.

Table 12. DB2 CLI Function List by Category (continued)

Task Function Name	ODBC 3.0	SQL/CLI	DB2 CLI First Version Supported	Purpose
SQLGetConnectOption()	Depr	95	V 2.1 [a]	Returns the value of a connection option.
SQLSetStmtAttr()	Core	95	V 5	Sets a statement attribute.
SQLSetStmtOption()	Depr	95	V 2.1 [a]	Sets a statement option.
SQLGetStmtAttr()	Core	95	V 5	Returns the value of a statement attribute.
SQLGetStmtOption()	Depr	95	V 2.1 [a]	Returns the value of a statement option.
Preparing SQL Requests				
SQLAllocStmt()	Depr	95	V 1.1	Allocates a statement handle.
SQLPrepare()	Core	95	V 1.1	Prepares an SQL statement for later execution.
SQLExtendedPrepare()	No	No	V 6	Prepares an array of statement attributes for an SQL statement for later execution.
SQLExtendedBind()	No	No	V 6	Bind an array of columns instead of using repeated calls to SQLBindCol() and SQLBindParameter()
SQLBindParameter()	Lvl 1	95 [b]	V 2.1	Assigns storage for a parameter in an SQL statement (ODBC 2.0)
SQLSetParam()	Depr	No	V 1.1	Assigns storage for a parameter in an SQL statement (ODBC 1.0). **Note:** In ODBC 2.0 this function has been replaced by SQLBindParameter().
SQLParamOptions()	Depr	No	V 2.1	Specifies the use of multiple values for parameters.
SQLGetCursorName()	Core	95	V 1.1	Returns the cursor name associated with a statement handle.
SQLSetCursorName()	Core	95	V 1.1	Specifies a cursor name.
Submitting Requests				
SQLDescribeParam()	Level 2	SQL3	V 5	Returns description of a parameter marker.
SQLExecute()	Core	95	V 1.1	Executes a prepared statement.
SQLExecDirect()	Core	95	V 1.1	Executes a statement.

Table 12. DB2 CLI Function List by Category (continued)

Task Function Name	ODBC 3.0	SQL/CLI	DB2 CLI First Version Supported	Purpose
SQLNativeSql()	Lvl 2	95	V 2.1 [a]	Returns the text of an SQL statement as translated by the driver.
SQLNumParams()	Lvl 2	95	V 2.1 [a]	Returns the number of parameters in a statement.
SQLParamData()	Lvl 1	95	V 2.1 [a]	Used in conjunction with SQLPutData() to supply parameter data at execution time. (Useful for long data values.)
SQLPutData()	Core	95	V 2.1 [a]	Send part or all of a data value for a parameter. (Useful for long data values.)
Retrieving Results and Information about Results				
SQLRowCount()	Core	95	V 1.1	Returns the number of rows affected by an insert, update, or delete request.
SQLNumResultCols()	Core	95	V 1.1	Returns the number of columns in the result set.
SQLDescribeCol()	Core	95	V 1.1	Describes a column in the result set.
SQLColAttribute()	Core	Yes	V 5	Describes attributes of a column in the result set.
SQLColAttributes()	Depr	Yes	V 1.1	Describes attributes of a column in the result set.
SQLColumnPrivileges()	Level 2	95	V 2.1	Get privileges associated with the columns of a table.
SQLSetColAttributes()	No	No	V 2.1	Sets attributes of a column in the result set.
SQLBindCol()	Core	95	V 1.1	Assigns storage for a result column and specifies the data type.
SQLFetch()	Core	95	V 1.1	Returns a result row.
SQLFetchScroll()	Core	95	V 5	Returns a rowset from a result row.
SQLExtendedFetch()	Depr	95	V 2.1	Returns multiple result rows.
SQLGetData()	Core	95	V 1.1	Returns part or all of one column of one row of a result set. (Useful for long data values.)

Table 12. DB2 CLI Function List by Category (continued)

Task Function Name	ODBC 3.0	SQL/CLI	DB2 CLI First Version Supported	Purpose
SQLMoreResults()	Lvl 1	SQL3	V 2.1 [a]	Determines whether there are more result sets available and, if so, initializes processing for the next result set.
SQLError()	Depr	95	V 1.1	Returns additional error or status information.
SQLGetDiagField()	Core	95	V 5	Get a field of diagnostic data.
SQLGetDiagRec()	Core	95	V 5	Get multiple fields of diagnostic data.
SQLSetPos()	Level 1	SQL3	V 5	Set the cursor position in a rowset.
SQLGetSQLCA()	No	No	V 2.1	Returns the SQLCA associated with a statement handle.
SQLBulkOperations()	Level 1	No	V 6	Perform bulk insertions, updates, deletions, and fetches by bookmark.
Descriptors				
SQLCopyDesc()	Core	95	V 5	Copy descriptor information between handles.
SQLGetDescField()	Core	95	V 5	Get single field settings of a descriptor record.
SQLGetDescRec()	Core	95	V 5	Get multiple field settings of a descriptor record.
SQLSetDescField()	Core	95	V 5	Set a single field of a descriptor record.
SQLSetDescRec()	Core	95	V 5	Set multiple field settings of a descriptor record.
Large Object Support				
SQLBindFileToCol()	No	No	V 2.1	Associates LOB file reference with a LOB column.
SQLBindFileToParam()	No	No	V 2.1	Associates LOB file reference with a parameter marker.
SQLGetLength()	No	SQL3	V 2.1	Gets length of a string referenced by a LOB locator.
SQLGetPosition()	No	SQL3	V 2.1	Gets the position of a string within a source string referenced by a LOB locator.

Table 12. DB2 CLI Function List by Category (continued)

Task Function Name	ODBC 3.0	SQL/CLI	DB2 CLI First Version Supported	Purpose
SQLGetSubString()	No	SQL3	V 2.1	Creates a new LOB locator that references a substring within a source string (the source string is also represented by a LOB locator).
Obtaining information about the data source's system tables (catalog functions)				
SQLColumns()	Lvl 1	SQL3	V 2.1 [a]	Returns the list of column names in specified tables.
SQLForeignKeys()	Lvl 2	SQL3	V 2.1	Returns a list of column names that comprise foreign keys, if they exist for a specified table.
SQLPrimaryKeys()	Lvl 1	SQL3	V 2.1	Returns the list of column name(s) that comprise the primary key for a table.
SQLProcedureColumns()	Lvl 2	No	V 2.1	Returns the list of input and output parameters for the specified procedures.
SQLProcedures()	Lvl 2	No	V 2.1	Returns the list of procedure names stored in a specific data source.
SQLSpecialColumns()	Core	SQL3	V 2.1 [a]	Returns information about the optimal set of columns that uniquely identifies a row in a specified table.
SQLStatistics()	Core	SQL3	V 2.1 [a]	Returns statistics about a single table and the list of indexes associated with the table.
SQLTablePrivileges()	Lvl 2	SQL3	V 2.1	Returns a list of tables and the privileges associated with each table.
SQLTables()	Core	SQL3	V 2.1 [a]	Returns the list of table names stored in a specific data source.
Terminating a Statement				
SQLFreeHandle()	Core	95	V 1.1	Free Handle Resources.
SQLFreeStmt()	Core	95	V 1.1	End statement processing and closes the associated cursor, discards pending results, and, optionally, frees all resources associated with the statement handle.
SQLCancel()	Core	95	V 1.1	Cancels an SQL statement.
SQLTransact()	Depr	No	V 1.1	Commits or rolls back a transaction.

Table 12. DB2 CLI Function List by Category (continued)

Task Function Name	ODBC 3.0	SQL/CLI	DB2 CLI First Version Supported	Purpose
SQLCloseCursor()	Core	95	V 5	Commits or rolls back a transaction.
Terminating a Connection				
SQLDisconnect()	Core	95	V 1.1	Closes the connection.
SQLEndTran()	Core	95	V 5	Ends transaction of a connection.
SQLFreeConnect()	Depr	95	V 1.1	Releases the connection handle.
SQLFreeEnv()	Depr	95	V 1.1	Releases the environment handle.

Note:

[a] Runtime support for this function was also available in the DB2 Client Application Enabler for DOS Version 1.2 product.

[b] SQLBindParam() has been replaced by SQLBindParameter().

The ODBC function(s):

- SQLSetScrollOptions is supported for runtime only, because it has been superceded by the SQL_CURSOR_TYPE, SQL_CONCURRENCY, SQL_KEYSET_SIZE, and SQL_ROWSET_SIZE statement options.
- SQLDrivers() is implemented by the ODBC driver manager.

SQLAllocConnect - Allocate Connection Handle

Deprecated

Note:

In ODBC version 3, SQLAllocConnect() has been deprecated and replaced with SQLAllocHandle(); see "SQLAllocHandle - Allocate Handle" on page 220 for more information.

Although this version of DB2 CLI continues to support SQLAllocConnect(), we recommend that you begin using SQLAllocHandle() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Migrating to the New Function

The statement:

```
SQLAllocConnect(henv, hdbc);
```

for example, would be rewritten using the new function as:

```
SQLAllocHandle(SQL_HANDLE_DBC, henv, hdbc);
```

SQLAllocEnv - Allocate Environment Handle

Usage

Note:

In ODBC version 3, SQLAllocEnv has been deprecated and replaced with SQLAllocHandle() see "SQLAllocHandle - Allocate Handle" on page 220 for more information.

Although this version of DB2 CLI continues to support SQLAllocEnv, we recommend that you begin using SQLAllocHandle() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Migrating to the New Function

The statement:

```
SQLAllocEnv(&henv);
```

for example, would be rewritten using the new function as:

```
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
```

SQLAllocHandle

SQLAllocHandle - Allocate Handle

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLAllocHandle() allocates environment, connection, statement, or descriptor handles.

Note: This function is a generic function for allocating handles that replaces the deprecated version 2 functions SQLAllocConnect(), SQLAllocEnv(), and SQLAllocStmt().

Syntax

```
SQLRETURN   SQLAllocHandle   (SQLSMALLINT       HandleType,
                              SQLHANDLE         InputHandle,
                              SQLHANDLE         *OutputHandlePtr);
```

Function Arguments

Table 13. SQLAllocHandle Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	HandleType	input	The type of handle to be allocated by SQLAllocHandle(). Must be one of the following values: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC
SQLHANDLE	InputHandle	input	Existing handle to use as a context for the new handle being allocated. If HandleType is SQL_HANDLE_ENV, this is SQL_NULL_HANDLE. If HandleType is SQL_HANDLE_DBC, this must be an environment handle, and if it is SQL_HANDLE_STMT or SQL_HANDLE_DESC, it must be a connection handle.
SQLHANDLE	OutputHandlePtr	output	Pointer to a buffer in which to return the handle to the newly allocated data structure.

Usage

SQLAllocHandle() is used to allocate environment, connection, statement, and descriptor handles, as described below.

Multiple environment, connection, or statement handles can be allocated by an application at a time.

If the application calls SQLAllocHandle() with *OutputHandlePtr set to an environment, connection, statement, or descriptor handle that already exists, DB2 CLI overwrites the information associated with the handle. DB2 CLI does not check to see whether the handle entered in *OutputHandlePtr is already in use, nor does it check the previous contents of a handle before overwriting them.

On operating systems that support multiple threads, applications can use the same environment, connection, statement, or descriptor handle on different threads. DB2 CLI provides thread safe access for all handles and function calls. The application itself might experience unpredictable behavior if the threads it creates do not co-ordinate their use of DB2 CLI resources. For more information refer to “Writing Multi-Threaded Applications” on page 46.

Allocating an Environment Handle

An environment handle provides access to global information such as valid connection handles and active connection handles. To request an environment handle, an application calls SQLAllocHandle() with a HandleType of SQL_HANDLE_ENV and a InputHandle of SQL_NULL_HANDLE. DB2 CLI allocates the environment handle, and passes the value of the associated handle back in *OutputHandlePtr argument. The application passes the *OutputHandle value in all subsequent calls that require an environment handle argument.

When DB2 CLI processes the SQLAllocHandle() function with a *HandleType* of SQL_HANDLE_ENV, it checks the **Trace** keyword in the [COMMON] section of the db2cli.ini file. If it is set to 1, DB2 CLI enables tracing for the current application. If the trace flag is set, tracing starts when the first environment handle is allocated, and ends when the last environment handle is freed. For more information, see “TRACE” on page 201.

After allocating an environment handle, an application should call SQLSetEnvAttr() on the environment handle to set the SQL_ATTR_ODBC_VERSION environment attribute. If the application is run as an ODBC application, and this attribute is not set before SQLAllocHandle() is called to allocate a connection handle on the environment, then the call to allocate the connection will return SQLSTATE HY010 (Function sequence error.).

Allocating a Connection Handle

A connection handle provides access to information such as the valid statement and descriptor handles on the connection and whether a transaction is currently open. To request a connection handle, an application calls SQLAllocHandle() with a HandleType of SQL_HANDLE_DBC. The InputHandle argument is set to the environment handle that was returned by the call to SQLAllocHandle() that allocated that handle. DB2 CLI allocates the connection handle, and passes the value of the associated handle back in *OutputHandlePtr. The application passes the *OutputHandlePtr value in all subsequent calls that require a connection handle.

If the SQL_ATTR_ODBC_VERSION environment attribute is not set before SQLAllocHandle() is called to allocate a connection handle on the environment, then the call to allocate the connection will return SQLSTATE HY010 (Function sequence error.) when the application is using the ODBC Driver Manager.

Allocating a Statement Handle

A statement handle provides access to statement information, such as error messages, the cursor name, and status information for SQL statement processing. To request a statement handle, an application connects to a data source, and then calls SQLAllocHandle() prior to submitting SQL statements. In this call, HandleType should be set to SQL_HANDLE_STMT and InputHandle should be set to the connection handle that was returned by the call to SQLAllocHandle() that allocated that handle. DB2 CLI allocates the statement handle, associates the statement handle with the connection specified, and passes the value of the associated handle back in *OutputHandlePtr. The application passes the *OutputHandlePtr value in all subsequent calls that require a statement handle.

When the statement handle is allocated, DB2 CLI automatically allocates a set of four descriptors, and assigns the handles for these descriptors to the SQL_ATTR_APP_ROW_DESC, SQL_ATTR_APP_PARAM_DESC, SQL_ATTR_IMP_ROW_DESC, SQL_ATTR_IMP_PARAM_DESC statement attributes. To use explicitly allocated application descriptors instead of the automatically allocated ones, see the “Allocating a Descriptor Handle” section below.

Allocating a Descriptor Handle

When an application calls SQLAllocHandle() with a HandleType of SQL_HANDLE_DESC, DB2 CLI allocates an application descriptor explicitly. The application can use an explicitly allocated application descriptor in place of an automatically allocated one by calling the SQLSetStmtAttr() function with the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC

attribute. An implementation descriptor cannot be allocated explicitly, nor can an implementation descriptor be specified in a SQLSetStmtAttr() function call.

Explicitly allocated descriptors are associated with a connection handle rather than a statement handle (as automatically allocated descriptors are). Descriptors can be associated with a connection handle only when an application is actually connected to the database. Since explicitly allocated descriptors are associated with a connection handle, an application can explicitly associate an allocated descriptor with more than one statement within a connection. An automatically allocated application descriptor, on the other hand, cannot be associated with more than one statement handle. Explicitly allocated descriptor handles can either be freed explicitly by the application, by calling SQLFreeHandle() with a HandleType of SQL_HANDLE_DESC, or freed implicitly when the connection handle is freed upon disconnect.

When an explicitly allocated application descriptor is associated with a statement, the automatically allocated descriptor that is no longer used is still associated with the connection handle. When the explicitly allocated descriptor is freed, the automatically allocated descriptor is once again associated with the statement (the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC Attribute for that statement is once again set to the automatically allocated descriptor handle). This is true for all statements that were associated with the explicitly allocated descriptor on the connection; each statement's original automatically allocated descriptor handle is again associated with that statement.

When a descriptor is first used, the initial value of its SQL_DESC_TYPE field is SQL_C_DEFAULT. DATA_PTR, INDICATOR_PTR, and OCTET_LENGTH_PTR are all initially set to null pointers. For the initial values of other fields, see "SQLSetDescField - Set a Single Field of a Descriptor Record" on page 649.

For more information see "Using Descriptors" on page 97.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

When allocating a handle other than an environment handle, if SQLAllocHandle() returns SQL_ERROR, it will set OutputHandlePtr to SQL_NULL_HENV, SQL_NULL_HDBC, SQL_NULL_HSTMT, or SQL_NULL_HDESC, depending on the value of HandleType, unless the

output argument is a null pointer. The application can then obtain additional information from the diagnostic data structure associated with the handle in the InputHandle argument.

Environment Handle Allocation Errors

Diagnostics

Table 14. SQLAllocHandle SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The HandleType argument was SQL_HANDLE_STMT or SQL_HANDLE_DESC, but the connection specified by the *InputHandle* argument was not open. The connection process must be completed successfully (and the connection must be open) for DB2 CLI to allocate a statement or descriptor handle.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory for the specified handle.
HY010	Function sequence error.	The *HandleType* argument was SQL_HANDLE_DBC, and `SQLSetEnvAttr()` has not been called to set the SQL_ODBC_VERSION environment attribute.
HY013	Unexpected memory handling error.	The HandleType argument was SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC; and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY014	No more handles.	The limit for the number of handles that can be allocated for the type of handle indicated by the *HandleType* argument has been reached.
HY092	Option type out of range.	The *HandleType* argument was not: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC
HYC00	Driver not capable.	The *HandleType* argument was SQL_HANDLE_DESC but the DB2 CLI driver was Version 2 or earlier.

Restrictions

None.

Example

Refer to the “Example” on page 326.

References

- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLExecute - Execute a Statement” on page 373
- “SQLFreeHandle - Free Handle Resources” on page 431
- “SQLPrepare - Prepare a Statement” on page 583
- “SQLSetConnectAttr - Set Connection Attributes” on page 618
- “SQLGetDescField - Get Single Field Settings of Descriptor Record” on page 458
- “SQLSetEnvAttr - Set Environment Attribute” on page 682
- “SQLSetStmtAttr - Set Options Related to a Statement” on page 702

SQLAllocStmt - Allocate a Statement Handle

Deprecated

Note:

In ODBC version 3, SQLAllocStmt() has been deprecated and replaced with SQLAllocHandle(); see "SQLAllocHandle - Allocate Handle" on page 220 for more information.

Although this version of DB2 CLI continues to support SQLAllocStmt(), we recommend that you begin using SQLAllocHandle() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Migrating to the New Function

The statement:

```
SQLAllocStmt(hdbc, &hstmt);
```

for example, would be rewritten using the new function as:

```
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
```

SQLBindCol - Bind a Column to an Application Variable or LOB Locator

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLBindCol() is used to associate (bind) columns in a result set to either:

- Application variables or arrays of application variables (storage buffers), for all C data types. In this case, data is transferred from the DBMS to the application when SQLFetch() or SQLFetchScroll() is called. Data conversion may occur as the data is transferred.
- A LOB locator, for LOB columns. In this case a LOB locator, not the data itself, is transferred from the DBMS to the application when SQLFetch() is called.

  Alternatively, LOB columns can be bound directly to a file using SQLBindFileToCol().

SQLBindCol() is called once for each column in the result set that the application needs to retrieve.

In general, SQLPrepare(), SQLExecDirect() or one of the schema functions is called before this function, and SQLFetch() or SQLFetchScroll() is called after. Column attributes may also be needed before calling SQLBindCol(), and can be obtained using SQLDescribeCol() or SQLColAttribute().

Syntax

```
SQLRETURN   SQLBindCol       (SQLHSTMT          StatementHandle,
                              SQLUSMALLINT      ColumnNumber,
                              SQLSMALLINT       TargetType,
                              SQLPOINTER        TargetValuePtr,
                              SQLINTEGER        BufferLength,
                              SQLINTEGER   *FAR StrLen_or_IndPtr);
```

Function Arguments

Table 15. SQLBindCol Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle

Table 15. SQLBindCol Arguments (continued)

Data Type	Argument	Use	Description
SQLUSMALLINT	*ColumnNumber*	input	Number identifying the column. Columns are numbered sequentially, from left to right. • Column numbers start at 1 if bookmarks are not used (SQL_ATTR_USE_BOOKMARKS statement attribute set to SQL_UB_OFF). • Column numbers start at 0 if bookmarks are used (the statement attribute set to SQL_UB_ON).
SQLSMALLINT	*TargetType*	input	The C data type for column number *ColumnNumber* in the result set. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_NUMERIC [a] • SQL_C_SBIGINT • SQL_C_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_UBIGINT Specifying SQL_C_DEFAULT causes data to be transferred to its default C data type, refer to Table 2 on page 29 for more information.

Table 15. SQLBindCol Arguments (continued)

Data Type	Argument	Use	Description
SQLPOINTER	*TargetValuePtr*	input/output (deferred)	Pointer to buffer (or an array of buffers if using SQLFetchScroll()) where DB2 CLI is to store the column data or the LOB locator when the fetch occurs. This buffer is used to return data when the *Operation* argument to SQLSetPos is SQL_REFRESH. The buffer is used to retrieve data when the SQLSetPos *Operation* argument is set to SQL_UPDATE. If TargetValuePtr is null, the column is unbound.
SQLINTEGER	*BufferLength*	input	Size of *TargetValuePtr* buffer in bytes available to store the column data or the LOB locator. If *TargetType* denotes a binary or character string (either single or double byte) or is SQL_C_DEFAULT, then BufferLength must be > 0, or an error will be returned. Otherwise, this argument is ignored.

Table 15. SQLBindCol Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER *	*StrLen_or_IndPtr*	input/output (deferred)	Pointer to value (or array of values) which indicates the number of bytes DB2 CLI has available to return in the *TargetValuePtr* buffer. If *TargetType* is a LOB locator, the size of the locator is returned, not the size of the LOB data. This buffer is used to return data when the *Operation* argument to SQLSetPos is SQL_REFRESH. The buffer is used to retrieve data when the SQLSetPos *Operation* argument is set to SQL_UPDATE. SQLFetch() returns SQL_NULL_DATA in this argument if the data value of the column is null. This pointer value must be unique for each bound column, or NULL. A value of SQL_COLUMN_IGNORE can also be set for use with SQLBulkOperations(). See "SQLBulkOperations - Add, Update, Delete or Fetch a Set of Rows" on page 276 for more details. SQL_NO_LENGTH may also be returned, refer to the Usage section below for more information.

- For this function, both *TargetValuePtr* and *StrLen_or_Ind* are deferred outputs, meaning that the storage locations these pointers point to do not get updated until a result set row is fetched. As a result, the locations referenced by these pointers must remain valid until SQLFetch() or SQLFetchScroll() is called. For example, if SQLBindCol() is called within a local function, SQLFetch() must be called from within the same scope of the function or the *TargetValuePtr* buffer must be allocated as static or global.
- DB2 CLI will be able to optimize data retrieval for all variable length data types if *TargetValuePtr* is placed consecutively in memory after *StrLen_or_IndPtr*, see below for more details.

Usage

The application calls SQLBindCol() once for each column in the result set for which it wishes to retrieve either the data, or optionally in the case of LOB columns, a LOB locator. Result sets are generated either by calling SQLExecute(), SQLExecDirect(), SQLGetTypeInfo(), or one of the catalog

functions. When SQLFetch() is called, the data in each of these *bound* columns is placed into the assigned location (given by the pointers *TargetValuePtr* and *StrLen_or_Ind*). If *TargetType* is a LOB locator, a locator value is returned, not the LOB data; the LOB locator references the entire data value in the LOB column.

SQLFetch() and SQLFetchScroll() can be used to retrieve multiple rows from the result set into an array. In this case, *TargetValuePtr* references an array. For more information, refer to "Retrieving a Result Set into an Array" on page 90 and "SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns" on page 408.

Columns are identified by a number, assigned sequentially from left to right.

- Column numbers start at 1 if bookmarks are not used (SQL_ATTR_USE_BOOKMARKS statement attribute set to SQL_UB_OFF).
- Column numbers start at 0 if bookmarks are used (the statement attribute set to SQL_UB_ON).

If you are going to use bookmarks you must first set the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_ON.

The number of columns in the result set can be determined by calling SQLNumResultCols() or by calling SQLColAttribute() with the *DescType* argument set to SQL_COLUMN_COUNT.

The application can query the attributes (such as data type and length) of the column by first calling SQLDescribeCol() or SQLColAttribute(). This information can then be used to allocate a storage location of the correct data type and length, to indicate data conversion to another data type, or in the case of LOB data types, optionally return a locator. Refer to "Data Types and Data Conversion" on page 28 for more information on default types and supported conversions.

An application can choose not to bind every column, or even not to bind any columns. Data in any of the columns can also be retrieved using SQLGetData() after the bound columns have been fetched for the current row. Generally, SQLBindCol() is more efficient than SQLGetData(). For a discussion of when to use one function over the other, refer to "Appendix A. Programming Hints and Tips" on page 757.

In subsequent fetches, the application can change the binding of these columns or bind previously unbound columns by calling SQLBindCol(). The new binding does not apply to data already fetched, it will be used on the next fetch. To unbind a single column (including columns bound with SQLBindFileToCol()), call SQLBindCol() with the *TargetValuePtr* pointer set to

NULL. To unbind all the columns, the application should call SQLFreeStmt() with the *Option* input set to SQL_UNBIND.

Instead of multiple calls to SQLBindCol(), DB2 CLI also supports column binding offsets. Rather than re-binding each time, an offset can be used to specify new buffer and length/indicator addresses which will be used in a subsequent call to SQLFetch() or SQLFetchScroll(). This can only be used with row wise binding, but will work whether the application retrieves a single row or multiple rows at a time.

See "Column Binding Offsets" on page 93 for the list of steps required to use an offset.

The application must ensure enough storage is allocated for the data to be retrieved. If the buffer is to contain variable length data, the application must allocate as much storage as the maximum length of the bound column requires; otherwise, the data may be truncated. If the buffer is to contain fixed length data, DB2 CLI assumes the size of the buffer is the length of the C data type. If data conversion is specified, the required size may be affected, see "Data Types and Data Conversion" on page 28 for more information.

If string truncation does occur, SQL_SUCCESS_WITH_INFO is returned and *StrLen_or_IndPtr* will be set to the actual size of *TargetValuePtr* available for return to the application.

Truncation is also affected by the SQL_ATTR_MAX_LENGTH statement attribute (used to limit the amount of data returned to the application). The application can specify not to report truncation by calling SQLSetStmtAttr() with SQL_ATTR_MAX_LENGTH and a value for the maximum length to return for all variable length columns, and by allocating a *TargetValuePtr* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS will be returned when the value is fetched and the maximum length, not the actual length, will be returned in *StrLen_or_IndPtr*.

If the column to be bound is a SQL_GRAPHIC, SQL_VARGRAPHIC or SQL_LONGVARGRAPHIC type, then *TargetType* can be set to SQL_C_DBCHAR or SQL_C_CHAR. If *TargetType* is SQL_C_DBCHAR, the data fetched into the *TargetValuePtr* buffer will be null-terminated with a double byte null-terminator. If *TargetType* is SQL_C_CHAR, then there will be no null-termination of the data. In both cases, the length of the *TargetValuePtr* buffer (*BufferLength*) is in units of bytes and should therefore be a multiple of 2. It is also possible to force DB2 CLI to null terminate graphic strings, see the PATCH1 keyword in "Configuration Keywords" on page 164.

Note: SQL_NO_TOTAL will be returned in *StrLen_or_IndPtr* if:

- The SQL type is a variable length type, and
- *StrLen_or_IndPtr* and *TargetValuePtr* are contiguous, and
- The column type is NOT NULLABLE, and
- String truncation occurred.

LOB locators can in general be treated as any other data type, but there are some important differences:

- Locators are generated at the server when a row is fetched and a LOB locator C data type is specified on SQLBindCol(), or when SQLGetSubString() is called to define a locator on a portion of another LOB. Only the locator is transferred to the application.
- The value of the locator is only valid within the current transaction. You cannot store a locator value and use it beyond the current transaction, even if the cursor used to fetch the LOB locator has the WITH HOLD attribute.
- A locator can also be freed before the end of the transaction with the FREE LOCATOR statement.
- Once a locator is received, the application can use SQLGetSubString(), to either receive a portion of the LOB value, or to generate another locator representing the sub-string. The locator value can also be used as input for a parameter marker (using SQLBindParameter()).

  A LOB locator is not a pointer to a database position, but rather it is a reference to a LOB value: a snapshot of that LOB value. There is no association between the current position of the cursor and the row from which the LOB value was extracted. This means that even after the cursor has moved to a different row, the LOB locator (and thus the value that it represents) can still be referenced.
- SQLGetPosition() and SQLGetLength() can be used with SQLGetSubString() to define the sub-string.

For a given LOB column in the result set, the binding can be to a:

- storage buffer for holding the entire LOB data value,
- LOB locator, or
- LOB file reference (using SQLBindFileToCol()).

The most recent bind column function call determines the type of binding that is in effect.

Descriptors and SQLBindCol

The following sections describe how SQLBindCol() interacts with descriptors.

Note: Calling SQLBindCol() for one statement can affect other statements. This occurs when the ARD associated with the statement is explicitly

allocated and is also associated with other statements. Because SQLBindCol() modifies the descriptor, the modifications apply to all statements with which this descriptor is associated. If this is not the required behavior, the application should dissociate this descriptor from the other statements before calling SQLBindCol().

Argument Mappings

Conceptually, SQLBindCol() performs the following steps in sequence:

1. Calls SQLGetStmtAttr() to obtain the ARD handle.
2. Calls SQLGetDescField() to get this descriptor's SQL_DESC_COUNT field, and if the value in the *ColumnNumber* argument exceeds the value of SQL_DESC_COUNT, calls SQLSetDescField() to increase the value of SQL_DESC_COUNT to *ColumnNumber*.
3. Calls SQLSetDescField() multiple times to assign values to the following fields of the ARD:
   - Sets SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE to the value of *TargetType*, except that if *TargetType* is one of the concise identifiers of a datetime or interval subtype, it sets SQL_DESC_TYPE to SQL_DATETIME or SQL_INTERVAL, respectively, sets SQL_DESC_CONCISE_TYPE to the concise identifier, and sets SQL_DESC_DATETIME_INTERVAL_CODE to the corresponding datetime or interval subcode.
   - Sets one or more of SQL_DESC_LENGTH, SQL_DESC_PRECISION, SQL_DESC_SCALE, and SQL_DESC_DATETIME_INTERVAL_PRECISION, as appropriate for TargetType.
   - Sets the SQL_DESC_OCTET_LENGTH field to the value of *BufferLength*.
   - Sets the SQL_DESC_DATA_PTR field to the value of *TargetValue*.
   - Sets the SQL_DESC_INDICATOR_PTR field to the value of *StrLen_or_Ind* (see the following paragraph).
   - Sets the SQL_DESC_OCTET_LENGTH_PTR field to the value of *StrLen_or_Ind* (see the following paragraph).

The variable that the *StrLen_or_Ind* argument refers to is used for both indicator and length information. If a fetch encounters a null value for the column, it stores SQL_NULL_DATA in this variable; otherwise, it stores the data length in this variable. Passing a null pointer as *StrLen_or_Ind* keeps the fetch operation from returning the data length, but makes the fetch fail if it encounters a null value and has no way to return SQL_NULL_DATA.

If the call to SQLBindCol() fails, the content of the descriptor fields it would have set in the ARD are undefined, and the value of the SQL_DESC_COUNT field of the ARD is unchanged.

Implicit Resetting of COUNT Field

SQLBindCol() sets SQL_DESC_COUNT to the value of the *ColumnNumber* argument only when this would increase the value of SQL_DESC_COUNT. If the value in the *TargetValuePtr* argument is a null pointer and the value in the *ColumnNumber* argument is equal to SQL_DESC_COUNT (that is, when unbinding the highest bound column), then SQL_DESC_COUNT is set to the number of the highest remaining bound column.

Cautions Regarding SQL_DEFAULT

To retrieve column data successfully, the application must determine correctly the length and starting point of the data in the application buffer. When the application specifies an explicit *TargetType*, application misconceptions are readily detected. However, when the application specifies a *TargetType* of SQL_DEFAULT, SQLBindCol() can be applied to a column of a different data type from the one intended by the application, either from changes to the metadata or by applying the code to a different column. In this case, the application may fail to determine the start or length of the fetched column data. This can lead to unreported data errors or memory violations.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 16. SQLBindCol SQLSTATEs

SQLSTATE	Description	Explanation
07009	Invalid descriptor index	The value specified for the argument *ColumnNumber* exceeded the maximum number of columns in the result set.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.

Table 16. SQLBindCol SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY002	Invalid column number.	The value specified for the argument *ColumnNumber* was less than 0. The value specified for the argument *ColumnNumber* exceeded the maximum number of columns supported by the data source.
HY003	Program type out of range.	*TargetType* was not a valid data type or SQL_C_DEFAULT.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* is less than 1 and the argument *TargetType* is either SQL_C_CHAR, SQL_C_BINARY or SQL_C_DEFAULT.
HYC00	Driver not capable.	DB2 CLI recognizes, but does not support the data type specified in the argument *TargetType* A LOB locator C data type was specified, but the connected server does not support LOB data types.

Note: Additional diagnostic messages relating to the bound columns may be reported at fetch time.

Restrictions

The LOB data support is only available when connected to a server that supports Large Object data types. If the application attempts to specify a LOB locator C data type, SQLSTATE **HY**C00 will be returned.

Example

Refer to “Example” on page 406.

References

- “SQLBindFileToCol - Bind LOB File Reference to LOB Column” on page 237
- “SQLFetch - Fetch Next Row” on page 396
- “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408

SQLBindFileToCol - Bind LOB File Reference to LOB Column

Purpose

Specification:	DB2 CLI 2.1		

`SQLBindFileToCol()` is used to associate (bind) a LOB column in a result set to a file reference or an array of file references. This enables data in that column to be transferred directly into a file when each row is fetched for the statement handle.

The LOB file reference arguments (file name, file name length, file reference options) refer to a file within the application's environment (on the client). Before fetching each row, the application must make sure that these variables contain the name of a file, the length of the file name, and a file option (new / overwrite / append). These values can be changed between each fetch.

Syntax

```
SQLRETURN   SQLBindFileToCol (SQLHSTMT          StatementHandle,   /* hstmt */
                              SQLUSMALLINT      ColumnNumber,      /* icol */
                              SQLCHAR      *FAR FileName,
                              SQLSMALLINT  *FAR FileNameLength,
                              SQLUINTEGER  *FAR FileOptions,
                              SQLSMALLINT       MaxFileNameLength,
                              SQLINTEGER   *FAR StringLength,
                              SQLINTEGER   *FAR IndicatorValue);
```

Function Arguments

Table 17. SQLBindFileToCol Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLUSMALLINT	*icol*	input	Number identifying the column. Columns are numbered sequentially, from left to right, starting at 1.
SQLPOINTER	FileName	input (deferred)	Pointer to the location that will contain the file name or an array of file names at the time of the next fetch using the *StatementHandle*. This is either the complete path name of the file(s) or a relative file name(s). If relative file name(s) are provided, they are appended to the current path of the running application. This pointer cannot be NULL.

Table 17. SQLBindFileToCol Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT *	FileNameLength	input (deferred)	Pointer to the location that will contain the length of the file name (or an array of lengths) at the time of the next fetch using the *StatementHandle*. If this pointer is NULL, then a length of SQL_NTS is assumed. The maximum value of the file name length is 255.
SQLUINTEGER *	FileOptions	input (deferred)	Pointer to the location that will contain the file option or (array of file options) to be used when writing the file at the time of the next fetch using the *StatementHandle*. The following *FileOptions* are supported: **SQL_FILE_CREATE** Create a new file. If a file by this name already exists, SQL_ERROR will be returned. **SQL_FILE_OVERWRITE** If the file already exists, overwrite it. Otherwise, create a new file. **SQL_FILE_APPEND** If the file already exists, append the data to it. Otherwise, create a new file. Only one option can be chosen per file, there is no default.
SQLSMALLINT	MaxFileNameLength	input	This specifies the length of the *FileName* buffer or, if the application uses `SQLFetchScroll()` to retrieve multiple rows for the LOB column, this specifies the length of each element in the *FileName* array.
SQLINTEGER *	StringLength	output (deferred)	Pointer to the location that contains the length (or array of lengths) in bytes of the LOB data that is returned. If this pointer is NULL, nothing is returned.
SQLINTEGER *	IndicatorValue	output (deferred)	Pointer to the location that contains an indicator value (or array of values).

Usage

The application calls SQLBindFileToCol() once for each column that should be transferred directly to a file when a row is fetched. LOB data is written directly to the file without any data conversion, and without appending null-terminators.

FileName, *FileNameLength*, and *FileOptions* must be set before each fetch. When SQLFetch() or SQLFetchScroll() is called, the data for any column which has been bound to a LOB file reference is written to the file or files pointed to by that file reference. Errors associated with the deferred input argument values of SQLBindFileToCol() are reported at fetch time. The LOB file reference, and the deferred *StringLength* and *IndicatorValue* output arguments are updated between fetch operations.

If SQLFetchScroll() is used to retrieve multiple rows for the LOB column, *FileName*, *FileNameLength*, and *FileOptions* point to an array of LOB file reference variables. In this case, *MaxFileNameLength* specifies the length of each element in the *FileName* array and is used by DB2 CLI to determine the location of each element in the *FileName* array. The contents of the array of file references must be valid at the time of the SQLFetchScroll() call. The *StringLength* and *IndicatorValue* pointers each point to an array whose elements are updated upon the SQLFetchScroll() call.

Using SQLFetchScroll(), multiple LOB values can be written to multiple files, or to the same file depending on the file names specified. If writing to the same file, the SQL_FILE_APPEND file option should be specified for each file name entry. Only column-wise binding of arrays of file references is supported with SQLFetchScroll().

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 18. SQLBindFileToCol SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.

Table 18. SQLBindFileToCol SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY002	Invalid column number.	The value specified for the argument *icol* was less than 1. The value specified for the argument *icol* exceeded the maximum number of columns supported by the data source.
HY009	Invalid argument value.	*FileName*, *StringLength* or *FileOptions* is a null pointer.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument *MaxFileNameLength* was less than 0.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.

Restrictions

This function is not available when connected to DB2 servers that do not support Large Object data types. Call SQLGetFunctions() with the function type set to SQL_API_SQLBINDFILETOCOL and check the *SupportedPtr* output argument to determine if the function is supported for the current connection.

Example

```
/* From CLI sample showpic.c */
/* ... */

    SQLCHAR * stmt1 =
       "select employee.empno, firstnme || lastname as name "
       "from employee, emp_photo "
       "where employee.empno = emp_photo.empno and photo_format = ?" ;

    SQLCHAR * stmt2 =
       "SELECT picture FROM emp_photo "
       "WHERE empno  = ? AND photo_format = ?" ;

/* ... */

    /* Get Employee Number */
    printf("Select a Employee Number from the list above\n");
    gets((char *)Empno.s);
```

```
    /* Execute statement 2 which selects the picturee blob */
    rc = SQLExecDirect(hstmt, stmt2, SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* Bind blob column to a file */
    rc = SQLBindFileToCol(hstmt, 1, FName, &FNLength, &FOption,
                          13, NULL, &FNInd);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* Set deferred FName argument to putput file name.  */
    switch (Photo_Format[0]) {
    case 'x':
        sprintf((char *)FName, "P%s.xwd", Empno.s, Photo_Format);
        sprintf(buffer, "xwud -in %s &", FName);
        printf("xwud will be used to display image\n");
        break;
    case 'b':
        sprintf((char *)FName, "P%s.bmp", Empno.s, Photo_Format);
        sprintf(buffer, "iconedit %s", FName);
#if !defined(DB2WIN)
        printf("iconedit will be used to display image\n");
#endif
        break;
    case 'g':
        sprintf((char *)FName, "P%s.gif", Empno.s, Photo_Format);
        printf("Will create file: %s\n", FName);
        break;
    default :
        sprintf((char *)FName, "P%s.pic", Empno.s, Photo_Format);
        printf("Unknown Format, will attempt to create file: %s \n", FName);
    }

    /* Fetch the blob column into the bound file */
    rc = SQLFetch(hstmt);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf("%s has been sucessfully Created\n", FName);

#if !defined(DB2WIN)
    /*If supported, execute local display tool with the file just created */
    if (buffer[0] != '\0' ) system(buffer);
#endif
```

References

- "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227
- "SQLFetch - Fetch Next Row" on page 396
- "SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns" on page 408
- "SQLBindFileToParam - Bind LOB File Reference to LOB Parameter" on page 242

SQLBindFileToParam - Bind LOB File Reference to LOB Parameter

Purpose

Specification:	DB2 CLI 2.1		

SQLBindFileToParam() is used to associate (bind) a parameter marker in an SQL statement to a file reference or an array of file references. This enables data from the file to be transferred directly into a LOB column when that statement is subsequently executed.

The LOB file reference arguments (file name, file name length, file reference options) refer to a file within the application's environment (on the client). Before calling SQLExecute() or SQLExecDirect(), the application must make sure that this information is available in the deferred input buffers. These values can be changed between SQLExecute() calls.

Syntax

```
SQLRETURN SQLBindFileToParam (SQLHSTMT          StatementHandle,
                              SQLUSMALLINT      TargetType,
                              SQLSMALLINT       DataType,
                              SQLCHAR      *FAR FileName,
                              SQLSMALLINT  *FAR FileNameLength,
                              SQLUINTEGER  *FAR FileOptions,
                              SQLSMALLINT       MaxFileNameLength,
                              SQLINTEGER   *FAR IndicatorValue);
```

Function Arguments

Table 19. SQLBindFileToParam Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLUSMALLINT	TargetType	input	Parameter marker number. Parameters are numbered sequentially, from left to right, starting at 1.
SQLSMALLINT	DataType	input	SQL Data Type of the column. The data type must be one of: • SQL_BLOB • SQL_CLOB • SQL_DBCLOB

Table 19. SQLBindFileToParam Arguments (continued)

Data Type	Argument	Use	Description
SQLCHAR *	FileName	input (deferred)	Pointer to the location that will contain the file name or an array of file names when the statement (*StatementHandle*) is executed. This is either the complete path name of the file or a relative file name. If a relative file name is provided, it is appended to the current path of the client process. This argument cannot be NULL.
SQLSMALLINT *	FileNameLength	input (deferred)	Pointer to the location that will contain the length of the file name (or an array of lengths) at the time of the next `SQLExecute()` or `SQLExecDirect()` using the *StatementHandle*. If this pointer is NULL, then a length of SQL_NTS is assumed. The maximum value of the file name length is 255.
SQLUINTEGER *	FileOptions	input (deferred)	Pointer to the location that will contain the file option (or an array of file options) to be used when reading the file. The location will be accessed when the statement (*StatementHandle*) is executed. Only one option is supported (and it must be specified): **SQL_FILE_READ** A regular file that can be opened, read and closed. (The length is computed when the file is opened) This pointer cannot be NULL.
SQLSMALLINT	MaxFileNameLength	input	This specifies the length of the *FileName* buffer. If the application calls `SQLParamOptions()` to specify multiple values for each parameter, this is the length of each element in the *FileName* array.
SQLINTEGER *	IndicatorValue	output (deferred)	Pointer to the location that contains an indicator value (or array of values), which is set to SQL_NULL_DATA if the data value of the parameter is to be null. It must be set to 0 (or the pointer can be set to null) when the data value is not null.

SQLBindFileToParam

Usage

The application calls SQLBindFileToParam() once for each parameter marker whose value should be obtained directly from a file when a statement is executed. Before the statement is executed, *FileName*, *FileNameLength*, and *FileOptions* values must be set. When the statement is executed, the data for any parameter which has been bound using SQLBindFIleToParam() is read from the referenced file and passed to the server.

If the application uses SQLParamOptions() to specify multiple values for each parameter, then *FileName*, *FileNameLength*, and *FileOptions* point to an array of LOB file reference variables. In this case, *MaxFileNameLength* specifies the length of each element in the *FileName* array and is used by DB2 CLI to determine the location of each element in the *FileName* array.

A LOB parameter marker can be associated with (bound to) an input file using SQLBindFileToParam(), or with a stored buffer using SQLBindParameter(). The most recent bind parameter function call determines the type of binding that is in effect.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 20. SQLBindFileToParam SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY004	SQL data type out of range.	The value specified for *DataType* was not a valid SQL type for this function call.
HY009	Invalid argument value.	*FileName*, *FileOptions FileNameLength*, is a null pointer.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.

Table 20. SQLBindFileToParam SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the input argument *MaxFileNameLength* was less than 0.
HY093	Invalid parameter number.	The value specified for *TargetType* was either less than 1 or greater than the maximum number of parameters supported.
HYC00	Driver not capable.	The server does not support Large Object data types.

Restrictions

This function is not available when connected to DB2 servers that do not support Large Object data types. Call SQLGetFunctions() with the function type set to SQL_API_SQLBINDFILETOPARAM and check the *SupportedPtr* output argument to determine if the function is supported for the current connection.

Example

```
/* From CLI sample picin.c */
/* ... */
    SQLCHAR * stmt =
    "INSERT INTO emp_photo (empno, photo_format, picture) VALUES (?, ?, ?)" ;
/* ... */

    /* Prepare the INSERT statement */

    rc = SQLPrepare( hstmt, stmt, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          7, 0, Empno, 7, NULL);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          11, 0,  Photo_Format, 11, NULL);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* Bind the Blob file to the parameter */
    rc = SQLBindFileToParam(hstmt, 3, SQL_BLOB, FName, &FNlength,
                            &FOption, 255, &FInd);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf("Enter the Employee Number:\n");
    gets((char *)Empno);
    while (1) {
        printf("Which Picture Format [xwd (XWindows), bitmap]?\n");
        gets((char *)Photo_Format);
```

```
    if (strcmp((char *)Photo_Format, "xwd") == 0 ||
        strcmp((char *)Photo_Format, "bitmap") == 0) break;
    printf("Invalid Format!\n");
}
printf("Enter the filename of the Photo of type %s\n", Photo_Format);
gets((char *)FName);

rc = SQLExecute( hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

References

- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247
- “SQLExecute - Execute a Statement” on page 373
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLParamOptions - Specify an Input Array for a Parameter” on page 579

SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator

Purpose

Specification:	DB2 CLI 2.1	ODBC 2.0	

SQLBindParameter() is used to associate (bind) parameter markers in an SQL statement to either:

- Application variables or arrays of application variables (storage buffers), for all C data types. In this case data is transferred from the application to the DBMS when SQLExecute() or SQLExecDirect() is called. Data conversion may occur as the data is transferred.
- A LOB locator, for SQL LOB data types. In this case a LOB locator value, not the LOB data itself, is transferred from the application to the server when the SQL statement is executed.

  Alternatively, LOB parameters can be bound directly to a file using SQLBindFileToParam().

This function must also be used to bind an application storage to a parameter of a stored procedure CALL statement where the parameter may be input, output or both. This function is essentially an extension of SQLSetParam().

Syntax

```
SQLRETURN  SQL_API SQLBindParameter(
                SQLHSTMT          StatementHandle,  /* hstmt */
                SQLUSMALLINT      ParameterNumber,  /* ipar */
                SQLSMALLINT       InputOutputType,  /* fParamType */
                SQLSMALLINT       ValueType,        /* fCType */
                SQLSMALLINT       ParameterType,    /* fSqlType */
                SQLUINTEGER       ColumnSize,       /* cbColDef */
                SQLSMALLINT       DecimalDigits,    /* ibScale */
                SQLPOINTER        ParameterValuePtr,/* rgbValue */
                SQLINTEGER        BufferLength,     /* cbValueMax */
                SQLINTEGER *FAR   StrLen_or_IndPtr);/* pcbValue */
```

Function Arguments

Table 21. SQLBindParameter Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement Handle
SQLUSMALLINT	ParameterNumber	input	Parameter marker number, ordered sequentially left to right, starting at 1.

Table 21. SQLBindParameter Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	InputOutputType	input	The type of parameter. The value of the SQL_DESC_PARAMETER_TYPE field of the IPD is also set to this argument. The supported types are: • SQL_PARAM_INPUT: The parameter marker is associated with an SQL statement that is not a stored procedure CALL; or, it marks an input parameter of the CALLed stored procedure. When the statement is executed, actual data value for the parameter is sent to the server: the *ParameterValuePtr* buffer must contain valid input data value(s); the *StrLen_or_IndPtr* buffer must contain the corresponding length value or SQL_NTS, SQL_NULL_DATA, or (if the value should be sent via `SQLParamData()` and `SQLPutData()`) SQL_DATA_AT_EXEC. • SQL_PARAM_INPUT_OUTPUT: The parameter marker is associated with an input/output parameter of the CALLed stored procedure. When the statement is executed, actual data value for the parameter is sent to the server: the *ParameterValuePtr* buffer must contain valid input data value(s); the *StrLen_or_IndPtr* buffer must contain the corresponding length value or SQL_NTS, SQL_NULL_DATA, or (if the value should be sent via `SQLParamData()` and `SQLPutData()`) SQL_DATA_AT_EXEC. • SQL_PARAM_OUTPUT: The parameter marker is associated with an output parameter of the CALLed stored procedure or the return value of the stored procedure. After the statement is executed, data for the output parameter is returned to the application buffer specified by *ParameterValuePtr* and *StrLen_or_IndPtr*, unless both are NULL pointers, in which case the output data is discarded. If an output parameter does not have a return value then *StrLen_or_IndPtr* is set to SQL_NULL_DATA.

Table 21. SQLBindParameter Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	ValueType	input	C data type of the parameter. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_NUMERIC [a] • SQL_C_SBIGINT • SQL_C_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_UBIGINT Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type to the type indicated in *ParameterType*. • **a** Windows 32-bit only

Table 21. SQLBindParameter Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	ParameterType	input	SQL Data Type of the parameter. The supported types are: • SQL_BIGINT • SQL_BINARY • SQL_BLOB • SQL_BLOB_LOCATOR • SQL_CHAR • SQL_CLOB • SQL_CLOB_LOCATOR • SQL_DBCLOB • SQL_DBCLOB_LOCATOR • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC **Note:** SQL_BLOB_LOCATOR, SQL_CLOB_LOCATOR, SQL_DBCLOB_LOCATOR are application related concepts and do not map to a data type for column definition during a CREATE TABLE statement.

Table 21. SQLBindParameter Arguments (continued)

Data Type	Argument	Use	Description
SQLUINTEGER	ColumnSize	input	Precision of the corresponding parameter marker. If *ParameterType* denotes: • A binary or single byte character string (e.g. SQL_CHAR, SQL_BLOB), this is the maximum length in bytes for this parameter marker. • A double byte character string (e.g. SQL_GRAPHIC), this is the maximum length in double-byte characters for this parameter. • SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision. • Otherwise, this argument is ignored. If the column size is not known in advance then the application can set this value to zero. See "ColumnSize Not Known in Advance" on page 258 for more information.
SQLSMALLINT	DecimalDigits	input	Scale of the corresponding parameter if *ParameterType* is SQL_DECIMAL or SQL_NUMERIC. If *ParameterType* is SQL_TYPE_TIMESTAMP, this is the number of digits to the right of the decimal point in the character representation of a timestamp (for example, the scale of yyyy-mm-dd hh:mm:ss.fff is 3). Other than for the *ParameterType* values mentioned here, *DecimalDigits* is ignored.

Table 21. SQLBindParameter Arguments (continued)

Data Type	Argument	Use	Description
SQLPOINTER	ParameterValuePtr	input (deferred) and / or output (deferred)	• On input (*InputOutputType* set to SQL_PARAM_INPUT, or SQL_PARAM_INPUT_OUTPUT): At execution time, if *StrLen_or_IndPtr* does not contain SQL_NULL_DATA or SQL_DATA_AT_EXEC, then *ParameterValuePtr* points to a buffer that contains the actual data for the parameter. If *StrLen_or_IndPtr* contains SQL_DATA_AT_EXEC, then *ParameterValuePtr* is an application-defined 32-bit value that is associated with this parameter. This 32-bit value is returned to the application via a subsequent `SQLParamData()` call. If `SQLParamOptions()` is called to specify multiple values for the parameter, then *ParameterValuePtr* is a pointer to a input buffer array of *BufferLength* bytes. • On output (*InputOutputType* set to SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT): *ParameterValuePtr* points to the buffer where the output parameter value of the stored procedure will be stored. If *InputOutputType* is set to SQL_PARAM_OUTPUT, and both *ParameterValuePtr* and *StrLen_or_IndPtr* are NULL pointers, then the output parameter value or the return value from the stored procedure call is discarded.

Table 21. SQLBindParameter Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER	BufferLength	input	For character and binary data, *BufferLength* specifies the length of the *ParameterValuePtr* buffer (if is treated as a single element) or the length of each element in the *ParameterValuePtr* array (if the application calls `SQLParamOptions()` to specify multiple values for each parameter). For non-character and non-binary data, this argument is ignored -- the length of the *ParameterValuePtr* buffer (if it is a single element) or the length of each element in the *ParameterValuePtr* array (if `SQLParamOptions()` is used to specify an array of values for each parameter) is assumed to be the length associated with the C data type. For output parameters, *BufferLength* is used to determine whether to truncate character or binary output data in the following manner: • For character data, if the number of bytes available to return is greater than or equal to *BufferLength*, the data in *ParameterValuePtr* is truncated to *BufferLength-1* bytes and is null-terminated (unless null-termination has been turned off). • For binary data, if the number of bytes available to return is greater than *BufferLength*, the data in *ParameterValuePtr* is truncated to *BufferLength* bytes.

Table 21. SQLBindParameter Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER *	StrLen_or_IndPtr	input (deferred) and/or output (deferred)	- If this is an input or input/output parameter: This is the pointer to the location which contains (when the statement is executed) the length of the parameter marker value stored at *ParameterValuePtr*. To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA. If *ValueType* is SQL_C_CHAR, this storage location must contain either the exact length of the data stored at *ParameterValuePtr*, or SQL_NTS if the contents at *ParameterValuePtr* is null-terminated. If *ValueType* indicates character data (explicitly, or implicitly using SQL_C_DEFAULT), and this pointer is set to NULL, it is assumed that the application will always provide a null-terminated string in *ParameterValuePtr*. This also implies that this parameter marker will never have a null value. If *ParameterType* denotes a graphic data type and the *ValueType* is SQL_C_CHAR, the pointer to *StrLen_or_IndPtr* can never be NULL and the contents of *StrLen_or_IndPtr* can never hold SQL_NTS. In general for graphic data types, this length should be the number of octets that the double byte data occupies; therefore, the length should always be a multiple of 2. In fact, if the length is odd, then an error will occur when the statement is executed. When `SQLExecute()` or `SQLExecDirect()` is called, and *StrLen_or_IndPtr* points to a value of SQL_DATA_AT_EXEC, the data for the parameter will be sent with `SQLPutData()`. This parameter is referred to as a **data-at-execution** parameter.

Table 21. SQLBindParameter Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER *	StrLen_or_IndPtr *(cont)*	input (deferred) and/or output (deferred)	If SQLParamOptions() is used to specify multiple values for each parameter, *StrLen_or_IndPtr* points to an array of SQLINTEGER values where each of the elements can be the number of bytes in the corresponding *ParameterValuePtr* element (excluding the null-terminator), or SQL_NULL_DATA. - If this is an output parameter (*InputOutputType* is set to SQL_PARAM_OUTPUT): This must be an output parameter or return value of a stored procedure CALL and points to one of the following, after the execution of the stored procedure: • number of bytes available to return in *ParameterValuePtr*, excluding the null-termination character. • SQL_NULL_DATA • SQL_NO_TOTAL if the number of bytes available to return cannot be determined.

Usage

A parameter marker is represented by a "?" character in an SQL statement and is used to indicate a position in the statement where an application supplied value is to be substituted when the statement is executed. This value can be obtained from:

- An application variable.

  SQLBindParameter() (or SQLSetParam()) is used to bind the application storage area to the parameter marker.

- A LOB value from the database server (by specifying a LOB locator).

  SQLBindParameter() (or SQLSetParam()) is used to bind a LOB locator to the parameter marker. The LOB value itself is supplied by the database server, so only the LOB locator is transferred between the database server and the application.

  An application can use a locator with SQLGetSubString(), SQLGetPosition() or SQLGetLength(). SQLGetSubString() can either return another locator, or the data itself. All locators remain valid until the end of the transaction in which they were created (even when the cursor moves to another row, or until it is freed using the FREE LOCATOR statement.

- A file (within the applications environment) containing a LOB value.

  SQLBindFileToParam() is used to bind a file to a LOB parameter marker. When SQLExecDirect() is executed, DB2 CLI will transfer the contents of the file directly to the database server.

The application must bind a variable to each parameter marker in the SQL statement before executing the SQL statement. For this function, *ParameterValuePtr* and *StrLen_or_IndPtr* are deferred arguments, the storage locations must be valid and contain input data values when the statement is executed. This means either keeping the SQLExecDirect() or SQLExecute() call in the same procedure scope as the SQLBindParameter() calls, or, these storage locations must be dynamically allocated or declared statically or globally.

SQLBindParameter() (or SQLSetParam()) can be called before SQLPrepare() if the columns in the result set are known; otherwise, the attributes of the result set can be obtained after the statement is prepared.

Parameter markers are referenced by number (*ColumnNumber*) and are numbered sequentially from left to right, starting at 1.

All parameters bound by this function remain in effect until one of the following:

- SQLFreeStmt() is called with the SQL_RESET_PARAMS option, or
- SQLFreeHandle() is called with *HandleType* set to SQL_HANDLE_STMT, or
- SQLBindParameter() is called again for the same parameter *ParameterNumber* number

After the SQL statement has been executed, and the results processed, the application may wish to reuse the statement handle to execute a different SQL statement. If the parameter marker specifications are different (number of parameters, length or type) then SQLFreeStmt() should be called with SQL_RESET_PARAMS to reset or clear the parameter bindings.

The C buffer data type given by *ValueType* must be compatible with the SQL data type indicated by *ParameterType*, or an error will occur.

An application can pass the value for a parameter either in the *ParameterValuePtr* buffer or with one or more calls to SQLPutData(). In latter case, these parameters are data-at-execution parameters. The application informs DB2 CLI of a data-at-execution parameter by placing the SQL_DATA_AT_EXEC value in the *StrLen_or_IndPtr* buffer. It sets the *ParameterValuePtr* input argument to a 32 bit value which will be returned on a subsequent SQLParamData() call and can be used to identify the parameter position.

Since the data in the variables referenced by *ParameterValuePtr* and *StrLen_or_IndPtr* is not verified until the statement is executed, data content or format errors are not detected or reported until SQLExecute() or SQLExecDirect() is called.

SQLBindParameter() essentially extends the capability of the SQLSetParam() function by providing a method of:

- Specifying whether a parameter is input, input / output, or output, necessary for proper handling of parameters for stored procedures.
- Specifying an array of input parameter values when SQLParamOptions() is used in conjunction with SQLBindParameter(). SQLSetParam() can still be used to bind single element application variables to parameter markers that are not part of a stored procedure CALL statement.

The *InputOutputType* argument specifies the type of the parameter. All parameters in the SQL statements that do not call procedures are input parameters. Parameters in stored procedure calls can be input, input/output, or output parameters. Even though the DB2 stored procedure argument convention typically implies that all procedure arguments are input/output, the application programmer may still choose to specify more exactly the input or output nature on the SQLBindParameter() to follow a more rigorous coding style.

- If an application cannot determine the type of a parameter in a procedure call, set *InputOutputType* to SQL_PARAM_INPUT; if the data source returns a value for the parameter, DB2 CLI discards it.
- If an application has marked a parameter as SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT and the data source does not return a value, DB2 CLI sets the *StrLen_or_IndPtr* buffer to SQL_NULL_DATA.
- If an application marks a parameter as SQL_PARAM_OUTPUT, data for the parameter is returned to the application after the CALL statement has been processed. If the *ParameterValuePtr* and *StrLen_or_IndPtr* arguments are both null pointers, DB2 CLI discards the output value. If the data source does not return a value for an output parameter, DB2 CLI sets the *StrLen_or_IndPtr* buffer to SQL_NULL_DATA.
- For this function, *ParameterValuePtr* and *StrLen_or_IndPtr* are deferred arguments. In the case where *InputOutputType* is set to SQL_PARAM_INPUT or SQL_PARAM_INPUT_OUTPUT, the storage locations must be valid and contain input data values when the statement is executed. This means either keeping the SQLExecDirect() or SQLExecute() call in the same procedure scope as the SQLBindParameter() calls, or, these storage locations must be dynamically allocated or statically / globally declared.

Similarly, if *InputOutputType* is set to SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT, the ParameterValuePtr and StrLen_or_IndPtr buffer locations must remain valid until the CALL statement has been executed.

For character and binary C data, the *BufferLength* argument specifies the length of the *ParameterValuePtr* buffer if it is a single element; or, if the application calls SQLParamOptions() to specify multiple values for each parameter, *BufferLength* is the length of *each* element in the *ParameterValuePtr* array, INCLUDING the null-terminator. If the application specifies multiple values, *BufferLength* is used to determine the location of values in the *ParameterValuePtr* array. For all other types of C data, the *BufferLength* argument is ignored.

An application can pass the value for a parameter either in the *ParameterValuePtr* buffer or with one or more calls to SQLPutData(). In latter case, these parameters are data-at-execution parameters. The application informs DB2 CLI of a data-at-execution parameter by placing the SQL_DATA_AT_EXEC value in the StrLen_or_IndPtr buffer. It sets the *ParameterValuePtr* input argument to a 32 bit value which will be returned on a subsequent SQLParamData() call and can be used to identify the parameter position.

When SQLBindParameter() is used to bind an application variable to an output parameter for a stored procedure, DB2 CLI can provide some performance enhancement if the *ParameterValuePtr* buffer is placed consecutively in memory after the *StrLen_or_IndPtr* buffer. For example:

```
struct {  SQLINTEGER  StrLen_or_IndPtr;
          SQLCHAR     ParameterValuePtr[MAX_BUFFER];
       } column;
```

A parameter can only be bound to either a file or a storage location, not both. The most recent bind parameter function call determines the bind that is in effect.

ColumnSize Not Known in Advance

When actual size of the target column or output parameter is not known the application may specify 0 for the length of the column. (*ColumnSize* set to 0).

In previous releases, DB2 CLI would use the maximum size for the column's datatype when *ColumnSize* was set to 0. In some cases this resulted in the allocation of unnecessarily large blocks of memory. As of version 6 this behavior has changed.

If the column's datatype is of fixed-length, the DB2 CLI driver will base the length from the datatype itself. However, setting *ColumnSize* to 0 means different things when the datatype is of type character, binary string or large object.

Input Parameter
A 0 *ColumnSize* means that DB2 CLI will use the actual data length of the input value - determined at the time the statement is executed - as the size of the column or the stored procedure parameter. DB2 CLI will perform necessary conversion using this size.

Output Parameter (Stored Procedures only)
A 0 *ColumnSize* means that DB2 CLI will use *BufferLength* as the parameter's size. Note that this means that the store procedure must not return more than *BufferLength* bytes of data or a truncation error will occur.

For Input-output parameter (Store Procedure only)
A 0 *ColumnSize* means that DB2 CLI will set both the input and output to *BufferLength* as the target parameter. This means that the input data will be converted to this new size if necessary before being sent to the stored procedure and at most *BufferLength* bytes of data is expected to be returned.

Setting *ColumnSize* to 0 is not recommended unless it is required; it causes DB2 CLI to perform uneccessary checking for the length of the data at run time.

Parameter Binding Offsets

When an application needs to change parameter bindings it can call `SQLBindParameter()` a second time. This will change the bound parameter buffer address and the corresponding length/indicator buffer address used.

Instead of multiple calls to `SQLBindParameter()`, DB2 CLI also supports parameter binding offsets. Rather than re-binding each time, an offset can be used to specify new buffer and length/indicator addresses which will be used in a subsequent call to `SQLExecute()` or `SQLExecDirect()`. This cannot be used with column wise array inserts, but will work whether the application binds parameters individually or using an array.

See "Parameter Binding Offsets" on page 87 for the list of steps required to use an offset.

Descriptors

How a parameter is bound is determined by fields of the APDs and IPDs. The arguments in SQLBindParameter are used to set those descriptor fields. The fields can also be set by the SQLSetDescField functions, although SQLBindParameter is more efficient to use because the application does not have to obtain a descriptor handle to call SQLBindParameter.

Note: Calling SQLBindParameter() for one statement can affect other statements. This occurs when the ARD associated with the statement is explicitly allocated and is also associated with other statements. Because SQLBindParameter() modifies the fields of the APD, the modifications apply to all statements with which this descriptor is associated. If this is not the required behavior, the application should dissociate this descriptor from the other statements before calling SQLBindParameter().

Conceptually, SQLBindParameter() performs the following steps in sequence:

1. Calls SQLGetStmtAttr() to obtain the APD handle.
2. Calls SQLGetDescField() to get the APD's SQL_DESC_COUNT field, and if the value of the ColumnNumber argument exceeds the value of SQL_DESC_COUNT, calls SQLSetDescField() to increase the value of SQL_DESC_COUNT to ColumnNumber.
3. Calls SQLSetDescField() multiple times to assign values to the following fields of the APD:
   - Sets SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE to the value of ValueType, except that if ValueType is one of the concise identifiers of a datetime or interval subtype, it sets SQL_DESC_TYPE to SQL_DATETIME or SQL_INTERVAL respectively, sets SQL_DESC_CONCISE_TYPE to the concise identifier, and sets SQL_DESC_DATETIME_INTERVAL_CODE to the corresponding datetime or interval subcode.
   - Sets the SQL_DESC_DATA_PTR field to the value of ParameterValue.
   - Sets the SQL_DESC_OCTET_LENGTH_PTR field to the value of StrLen_or_Ind.
   - Sets the SQL_DESC_INDICATOR_PTR field also to the value of StrLen_or_Ind.

   The StrLen_or_Ind parameter specifies both the indicator information and the length for the parameter value.
4. Calls SQLGetStmtAttr() to obtain the IPD handle.
5. Calls SQLGetDescField() to get the IPD's SQL_DESC_COUNT field, and if the value of the ColumnNumber argument exceeds the value of SQL_DESC_COUNT, calls SQLSetDescField to increase the value of SQL_DESC_COUNT to ColumnNumber.

6. Calls SQLSetDescField() multiple times to assign values to the following fields of the IPD:
   - Sets SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE to the value of ParameterType, except that if ParameterType is one of the concise identifiers of a datetime or interval subtype, it sets SQL_DESC_TYPE to SQL_DATETIME or SQL_INTERVAL respectively, sets SQL_DESC_CONCISE_TYPE to the concise identifier, and sets SQL_DESC_DATETIME_INTERVAL_CODE to the corresponding datetime or interval subcode.
   - Sets one or more of SQL_DESC_LENGTH, SQL_DESC_PRECISION, and SQL_DESC_DATETIME_INTERVAL_PRECISION, as appropriate for ParameterType.
   - Sets SQL_DESC_SCALE to the value of DecimalDigits.

If the call to SQLBindParameter() fails, the content of the descriptor fields that it would have set in the APD are undefined, and the SQL_DESC_COUNT field of the APD is unchanged. In addition, the SQL_DESC_LENGTH, SQL_DESC_PRECISION, SQL_DESC_SCALE, and SQL_DESC_TYPE fields of the appropriate record in the IPD are undefined and the SQL_DESC_COUNT field of the IPD is unchanged.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 22. SQLBindParameter SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The conversion from the data value identified by the *ValueType* argument to the data type identified by the *ParameterType* argument is not a meaningful conversion. (For example, conversion from SQL_C_DATE to SQL_DOUBLE.)
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	The value specified by the argument *ParameterNumber* not a valid data type or SQL_C_DEFAULT.

Table 22. SQLBindParameter SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY004	SQL data type out of range.	The value specified for the argument *ParameterType* is not a valid SQL data type.
HY009	Invalid argument value.	The argument *ParameterValuePtr* was a null pointer and the argument *StrLen_or_IndPtr* was a null pointer, and *InputOutputType* is not SQL_PARAM_OUTPUT.
HY010	Function sequence error.	Function was called after `SQLExecute()` or `SQLExecDirect()` had returned SQL_NEED_DATA, but data have not been sent for all *data-at-execution* parameters.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY021	Inconsistent descriptor information	The descriptor information checked during a consistency check was not consistent.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* was less than 0.
HY093	Invalid parameter number.	The value specified for the argument *ValueType* was less than 1 or greater than the maximum number of parameters supported by the server.
HY094	Invalid scale value.	The value specified for *ParameterType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified for *DecimalDigits* was less than 0 or greater than the value for the argument *ParamDef* (precision). The value specified for *ParameterType* was SQL_C_TIMESTAMP and the value for *ParameterType* was either SQL_CHAR or SQL_VARCHAR and the value for *DecimalDigits* was less than 0 or greater than 6.
HY104	Invalid precision value.	The value specified for *ParameterType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified for *ParamDef* was less than 1.
HY105	Invalid parameter type.	*InputOutputType* is not one of SQL_PARAM_INPUT, SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT.
HYC00	Driver not capable.	DB2 CLI or data source does not support the conversion specified by the combination of the value specified for the argument *ValueType* and the value specified for the argument *ParameterType*. The value specified for the argument *ParameterType* is not supported by either DB2 CLI or the data source.

Restrictions

In DB2 CLI v5 and ODBC 2.0, this function has replaced `SQLSetParam()`.

A new value for *StrLen_or_IndPtr*, SQL_DEFAULT_PARAM, was introduced in ODBC 2.0, to indicate that the procedure is to use the default value of a parameter, rather than a value sent from the application. Since DB2 stored procedure arguments do not have the concept of default values, specification of this value for *StrLen_or_IndPtr* argument will result in an error when the CALL statement is executed since the SQL_DEFAULT_PARAM value will be considered an invalid length.

ODBC 2.0 also introduced the SQL_LEN_DATA_AT_EXEC(*length*) macro to be used with the *StrLen_or_IndPtr* argument. The macro is used to specify the sum total length of the entire data that would be sent for character or binary C data via the subsequent SQLPutData() calls. Since the DB2 ODBC driver does not need this information, the macro is not needed. An ODBC application calls SQLGetInfo() with the SQL_NEED_LONG_DATA_LEN option to check if the driver needs this information. The DB2 ODBC driver will return 'N' to indicate that this information is not needed by SQLPutData().

Example

The example shown below binds a variety of data types bound to a set of parameters. For an additional example refer to "Stored Procedure Example" on page 131.

```
/* From CLI sample prodin.c */
/* ... */
    SQLCHAR * stmt =
       "INSERT INTO PRODUCT VALUES (?, ?, ?, ?, ?)" ;

    SQLINTEGER Prod_Num[] = {
        100110, 100120, 100210, 100220, 100510, 100520, 200110,
        200120, 200210, 200220, 200510, 200610, 990110, 990120,
        500110, 500210, 300100
    };

    SQLCHAR * Description[] = {
        "Aquarium-Glass-25 litres", "Aquarium-Glass-50 litres",
        "Aquarium-Acrylic-25 litres", "Aquarium-Acrylic-50 litres",
        "Aquarium-Stand-Small", "Aquarium-Stand-Large",
        "Pump-Basic-25 litre", "Pump-Basic-50 litre",
        "Pump-Deluxe-25 litre", "Pump-Deluxe-50 litre",
        "Pump-Filter-(for Basic Pump)",
        "Pump-Filter-(for Deluxe Pump)",
        "Aquarium-Kit-Small", "Aquarium-Kit-Large",
        "Gravel-Colored", "Fish-Food-Deluxe-Bulk",
        "Plastic-Tubing"
    };

    SQLDOUBLE UPrice[] = {
        110.00, 190.00, 100.00, 150.00, 60.00, 90.00, 30.00,
        45.00, 55.00, 75.00, 4.75, 5.25, 160.00, 240.00,
```

```
            2.50, 35.00, 5.50
        };

        SQLCHAR * Units[] = {
            "  ", "  ", "  ", "  ", "  ", "  ", "  ", "  ", "  ", "  ",
            "  ", "  ", "  ", "  ", "  ", "kg", "kg", "m"
        };

        SQLCHAR * Combo[] = {
            "N", "N", "N", "N", "N", "N", "N", "N", "N",
            "N", "N", "N", "Y", "Y", "N", "N", "N"
        };

    /* ... */

        /* Prepare the statement */
        rc = SQLPrepare( hstmt, stmt, SQL_NTS ) ;
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

        rc = SQLSetStmtAttr( hstmt,
                             SQL_ATTR_PARAMSET_SIZE,
                             ( SQLPOINTER ) row_array_size,
                             0
                           ) ;
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

        rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER,
                              0, 0, Prod_Num, 0, NULL);
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

        rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
                         257, 0, Description, 257, NULL);
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

        rc = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_DOUBLE, SQL_DECIMAL,
                              10, 2, UPrice, 0, NULL);
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

        rc = SQLBindParameter(hstmt, 4, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                              3, 0, Units, 3, NULL);
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

        rc = SQLBindParameter(hstmt, 5, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                              2, 0, Combo, 2, NULL);
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

        rc = SQLExecute( hstmt ) ;
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
        printf( "Inserted %ld Rows\n", row_array_size ) ;
```

References

- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLExecute - Execute a Statement” on page 373

- "SQLParamData - Get Next Parameter For Which A Data Value Is Needed" on page 575
- "SQLParamOptions - Specify an Input Array for a Parameter" on page 579
- "SQLPutData - Passing Data Value for A Parameter" on page 609

SQLBrowseConnect - Get Required Attributes to Connect to Data source

Purpose

Specification:	DB2 CLI 5.0	ODBC 1	

SQLBrowseConnect() supports an iterative method of discovering and enumerating the attributes and attribute values required to connect to a data source. Each call to SQLBrowseConnect() returns successive levels of attributes and attribute values. When all levels have been enumerated, a connection to the data source is completed and a complete connection string is returned by SQLBrowseConnect(). A return code of SQL_SUCCESS or SQL_SUCCESS_WITH_INFO indicates that all connection information has been specified and the application is now connected to the data source.

Syntax

```
SQLRETURN   SQLBrowseConnect (SQLHDBC           ConnectionHandle,
                              SQLCHAR           *InConnectionString,
                              SQLSMALLINT       StringLength1,
                              SQLCHAR           *OutConnectionString,
                              SQLSMALLINT       BufferLength,
                              SQLSMALLINT       *StringLength2Ptr);
```

Function Arguments

Table 23. SQLBrowseConnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	ConnectionHandle	input	Connection handle.
SQLCHAR	*szConnStrIn	input	Browse request connection string (see “InConnectionString Argument” on page 267).
SQLSMALLINT	cbConnStrIn	input	Length of *InConnectionString.
SQLCHAR	*OutConnectionString	output	Pointer to a buffer in which to return the browse result connection string (see “OutConnectionString Argument” on page 267).
SQLINTEGER	BufferLength	input	Length of the *OutConnectionString buffer.
SQLSMALLINT	*StringLength2Ptr	output	The total number of bytes (excluding the null termination byte) available to return in *OutConnectionString. If the number of bytes available to return is greater than or equal to BufferLength, the connection string in *OutConnectionString is truncated to BufferLength minus the length of a null termination character.

Usage

InConnectionString Argument

A browse request connection string has the following syntax:

connection-string ::= attribute[;] | attribute; connection-string

attribute ::= attribute-keyword=attribute-value | DRIVER=[{]attribute-value[}]

attribute-keyword ::= DSN | UID | PWD | NEWPWD
| driver-defined-attribute-keyword

attribute-value ::= character-string
driver-defined-attribute-keyword ::= identifier

where

- character-string has zero or more characters
- identifier has one or more characters
- attribute-keyword is case insensitive
- attribute-value may be case sensitive
- the value of the **DSN** keyword does not consist solely of blanks
- NEWPWD is used as part of a change password request. The application can either specify the new string to use, for example, NEWPWD=anewpass; or specify NEWPWD=; and rely on a dialog box generated by the DB2 CLI driver to prompt for the new password

Because of connection string and initialization file grammar, keywords and attribute values that contain the characters **[]{}(),;?*=!@** should be avoided. Because of the grammar in the system information, keywords and data source names cannot contain the backslash (\) character. For DB2 CLI Version 2, braces are required around the DRIVER keyword.

If any keywords are repeated in the browse request connection string, DB2 CLI uses the value associated with the first occurrence of the keyword. If the **DSN** and **DRIVER** keywords are included in the same browse request connection string, DB2 CLI uses which ever keyword appears first.

OutConnectionString Argument

The browse result connection string is a list of connection attributes. A connection attribute consists of an attribute keyword and a corresponding attribute value. The browse result connection string has the following syntax:

connection-string ::= attribute[;] | attribute; connection-string

attribute ::= [*]attribute-keyword=attribute-value
attribute-keyword ::= ODBC-attribute-keyword
| driver-defined-attribute-keyword

ODBC-attribute-keyword = {UID | PWD}[:localized-identifier]
driver-defined-attribute-keyword ::= identifer[:localized-identifier]

attribute-value ::= {attribute-value-list} | ?
(The braces are literal; they are returned by DB2 CLI.)
attribute-value-list ::= character-string [:localized-character
string] | character-string [:localized-character string], attribute-value-list

where

- character-string and localized-character string have zero or more characters
- identifier and localized-identifier have one or more characters; attribute-keyword is case insensitive
- attribute-value may be case sensitive

Because of connection string and initialization file grammar, keywords, localized identifiers, and attribute values that contain the characters **[]{}(),;?*=!@** should be avoided. Because of the grammar in the system information, keywords and data source names cannot contain the backslash (\) character.

The browse result connection string syntax is used according to the following semantic rules:

- If an asterisk (*) precedes an attribute-keyword, the attribute is optional, and may be omitted in the next call to `SQLBrowseConnect()`.
- The attribute keywords **UID** and **PWD** have the same meaning as defined in `SQLDriverConnect()`.
- When connecting to a DB2 Universal Database, only **DSN**, **UID** and **PWD** are required. Other keywords can be specified but do not affect the connection.
- ODBC-attribute-keywords and driver-defined-attribute-keywords include a localized or user-friendly version of the keyword. This might be used by applications as a label in a dialog box. However, **UID**, **PWD**, or the identifier alone must be used when passing a browse request string to DB2 CLI.
- The {attribute-value-list} is an enumeration of actual values valid for the corresponding attribute-keyword. Note that the braces ({}) do not indicate a list of choices; they are returned by DB2 CLI. For example, it might be a list of server names or a list of database names.

- If the attribute-value is a single question mark (?), a single value corresponds to the attribute-keyword. For example, UID=JohnS; PWD=Sesame.
- Each call to SQLBrowseConnect() returns only the information required to satisfy the next level of the connection process. DB2 CLI associates state information with the connection handle so that the context can always be determined on each call.

Using SQLBrowseConnect

SQLBrowseConnect() requires an allocated connection. If SQLBrowseConnect() returns SQL_ERROR, outstanding connections are terminated and the connection is returned to an unconnected state.

When SQLBrowseConnect() is called for the first time on a connection, the browse request connection string must contain the DSN keyword.

On each call to SQLBrowseConnect(), the application specifies the connection attribute values in the browse request connection string. DB2 CLI returns successive levels of attributes and attribute values in the browse result connection string; it returns SQL_NEED_DATA as long as there are connection attributes that have not yet been enumerated in the browse request connection string. The application uses the contents of the browse result connection string to build the browse request connection string for the next call to SQLBrowseConnect(). All mandatory attributes (those not preceded by an asterisk in the *OutConnectionString* argument) must be included in the next call to SQLBrowseConnect(). Note that the application cannot use the contents of previous browse result connection strings when building the current browse request connection string; that is, it cannot specify different values for attributes set in previous levels.

When all levels of connection and their associated attributes have been enumerated, DB2 CLI returns SQL_SUCCESS, the connection to the data source is complete, and a complete connection string is returned to the application. The connection string is suitable to use in conjunction with SQLDriverConnect() with the SQL_DRIVER_NOPROMPT option to establish another connection. The complete connection string cannot be used in another call to SQLBrowseConnect(), however; if SQLBrowseConnect() were called again, the entire sequence of calls would have to be repeated.

SQLBrowseConnect() also returns SQL_NEED_DATA if there are recoverable, nonfatal errors during the browse process, for example, an invalid password supplied by the application or an invalid attribute keyword supplied by the application. When SQL_NEED_DATA is returned and the browse result connection string is unchanged, an error has occurred and the application can

call SQLGetDiagRec() to return the SQLSTATE for browse-time errors. This permits the application to correct the attribute and continue the browse.

An application may terminate the browse process at any time by calling SQLDisconnect(). DB2 CLI will terminate any outstanding connections and return the connection to an unconnected state.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 24. SQLBrowseConnect SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The buffer **OutConnectionString* was not large enough to return entire browse result connection string, so the string was truncated. The buffer **StringLength2Ptr* contains the length of the untruncated browse result connection string. (Function returns SQL_SUCCESS_WITH_INFO.)
01S00	Invalid connection string attribute.	An invalid attribute keyword was specified in the browse request connection string (*InConnectionString*). (Function returns SQL_NEED_DATA.) An attribute keyword was specified in the browse request connection string (*InConnectionString*) that does not apply to the current connection level. (Function returns SQL_NEED_DATA.)
01S02	Option value changed.	DB2 CLI did not support the specified value of the *ValuePtr* argument in SQLSetConnectAttr() and substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
08001	Unable to connect to data source.	DB2 CLI was unable to establish a connection with the data source.
08002	Connection in use.	The specified connection had already been used to establish a connection with a data source and the connection was open.

Table 24. SQLBrowseConnect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
08004	The application server rejected establishment of the connection.	The data source rejected the establishment of the connection for implementation defined reasons.
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was trying trying to connect failed before the function completed processing.
28000	Invalid authorization specification.	Either the user identifier or the authorization string or both as specified in the browse request connection string (*InConnectionString*) violated restrictions defined by the data source.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument *StringLength1* was less than 0 and was not equal to SQL_NTS. The value specified for argument *BufferLength* was less than 0.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- “SQLAllocHandle - Allocate Handle” on page 220
- “SQLConnect - Connect to a Data Source” on page 323
- “SQLDisconnect - Disconnect from a Data Source” on page 347
- “SQLDriverConnect - (Expanded) Connect to a Data Source” on page 350
- “SQLFreeHandle - Free Handle Resources” on page 431

SQLBuildDataLink

SQLBuildDataLink - Build DATALINK Value

Purpose

Specification:	DB2 CLI 5.2		ISO CLI

SQLBuildDataLink() returns a DATALINK value built from input arguments.

Syntax

```
SQLRETURN   SQLBuildDataLink(SQLHSTMT          StatementHandle,
                             SQLCHAR     FAR   *LinkType,
                             SQLINTEGER        LinkTypeLength,
                             SQLCHAR     FAR   *DataLocation,
                             SQLINTEGER        DataLocationLength,
                             SQLCHAR     FAR   *Comment,
                             SQLINTEGER        CommentLength,
                             SQLCHAR     FAR   *DataLinkValue,
                             SQLINTEGER        BufferLength,
                             SQLINTEGER  FAR   *StringLengthPtr);
```

Function Arguments

Table 25. SQLBuildDataLink Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Used only for diagnostic reporting.
SQLCHAR *	LinkType	input	Always set to SQL_DATALINK_URL.
SQLINTEGER	LinkTypeLength	input	The length of the *LinkType* value.
SQLCHAR *	DataLocation	input	The complete URL value to be assigned.
SQLINTEGER	DataLocationLength	input	The length of the *DataLocation* value.
SQLCHAR *	Comment	input	The comment, if any, to be assigned.
SQLINTEGER	CommentLength	input	The length of the *Comment* value.
SQLCHAR *	DataLinkValue	output	The DATALINK value that is created by the function.
SQLINTEGER	BufferLength	input	Length of the DataLinkValue buffer.
SQLINTEGER	*StringLengthPtr	output	A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character) available to return in **DataLinkValue*. If *DataLinkValue* is a null pointer, no length is returned. If the number of bytes available to return is greater than BufferLength minus the length of the null-termination character, then SQLSTATE 01004 is returned. In this case, subsequent use of the DATALINK value may fail.

Usage

The function is used to build a DATALINK value. The maximum length of the string, including the null termination character, will be *BufferLength* bytes.

Refer to the *Administration Guide, Design and Implementation* for more information on Data Links.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 26. SQLBuildDataLink() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
01004	Data truncated.	The data returned in **DataLinkValue* was truncated to be *BufferLength* minus the length of the null termination character. The length of the untruncated string value is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified one of the arguments (*LinkTypeLength*, *DataLocationLength*, or *CommentLength*) was less than 0 but not equal to SQL_NTS or *BufferLength* is less than 0.

Restrictions

None.

Example

```
/* ... */
void getattr( SQLHSTMT hStmt,
              SQLSMALLINT AttrType,
```

```
                SQLCHAR* DataLink,
                SQLCHAR* Attribute,
                SQLINTEGER BufferLength)
{
    SQLINTEGER  StringLength ;
    SQLRETURN   rc ;

    rc = SQLGetDataLinkAttr(
                              hStmt,
                              AttrType,
                              DataLink,
                              strlen( (char *)DataLink),
                              Attribute,
                              BufferLength,
                             &StringLength
                            ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hStmt, rc ) ;

    printf("Attribute #%d) >%s>\n", AttrType, Attribute) ;
    return ;
}

/* ... */

    SQLCHAR szCreate[] = "CREATE TABLE DL_SAMPL "
                                        "( "
                                        "DL1 DATALINK "
                                           "LINKTYPE URL "
                                           "FILE LINK CONTROL "
                                           "INTEGRITY ALL "
                                           "READ PERMISSION DB "
                                           "WRITE PERMISSION BLOCKED "
                                           "RECOVERY NO "
                                           "ON UNLINK RESTORE "
                                        ")";

    SQLCHAR szInsert[] = "INSERT INTO DL_SAMPL VALUES (?)" ;

    SQLCHAR szFileLink[] =
         "http://fearless.torolab.ibm.com/nfsdlink/rpomeroy/test_1.jpg" ;
    SQLCHAR szComment[] = "My First Datalink" ;

    SQLCHAR szSelect[] = "SELECT * FROM DL_SAMPL" ;
    SQLCHAR szDrop[] = "DROP TABLE DL_SAMPL" ;
    SQLCHAR szDLCol[254] ;
    SQLCHAR szBuffer[254] ;
    SQLSMALLINT cCol ;
    SQLCHAR szColName[33] ;
    SQLSMALLINT fSqlType ;
    SQLUINTEGER cbColDef ;
    SQLSMALLINT ibScale ;
    SQLSMALLINT fNullable ;
    SQLINTEGER  siLength = SQL_NTS ;

/* ... */
```

```
    /* Build Datalink */
    rc = SQLBuildDataLink( hstmt,
                           (SQLCHAR *)"URL",
                           strlen("URL"),
                           szFileLink,
                           strlen((char *)szFileLink),
                           szComment,
                           strlen((char *)szComment),
                           szDLCol,
                           sizeof(szDLCol),
                          &siLength
                         );
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /*  Set input parameter.  */
    rc = SQLBindParameter(
                           hstmt,
                           1,
                           SQL_PARAM_INPUT,
                           SQL_C_DATALINK,
                           SQL_DATALINK,
                           sizeof(szDLCol),
                           0,
                           (SQLPOINTER)szDLCol,
                           sizeof(szDLCol),
                           NULL
                         ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* ... */
}                               /* end main */
```

References

- "SQLGetDataLinkAttr - Get DataLink Attribute Value" on page 455

SQLBulkOperations - Add, Update, Delete or Fetch a Set of Rows

Purpose

Specification:	DB2 CLI 6.0	ODBC 3.0	

SQLBulkOperations() is used to perform the following operations on a keyset driven cursor:

- Add new rows
- Update a set of rows where each row is identified by a bookmark
- Delete a set of rows where each row is identified by a bookmark
- Fetch a set of rows where each row is identified by a bookmark

Syntax

```
SQLRETURN   SQLBulkOperations    (
                SQLHSTMT          StatementHandle,
                SQLSMALLINT       Operation);
```

Function Arguments

Table 27. SQLBulkOperations Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.
SQLSMALLINT	Operation	Input	Operation to perform: • SQL_ADD • SQL_UPDATE_BY_BOOKMARK • SQL_DELETE_BY_BOOKMARK • SQL_FETCH_BY_BOOKMARK

Usage

An application uses SQLBulkOperations() to perform the following operations on the base table or view that corresponds to the current query in a keyset driven cursor:

- Add new rows
- Update a set of rows where each row is identified by a bookmark
- Delete a set of rows where each row is identified by a bookmark
- Fetch a set of rows where each row is identified by a bookmark

A generic application should first ensure that the required bulk operation is supported. To do so, it can call SQLGetInfo() with an *InfoType* of SQL_DYNAMIC_CURSOR_ATTRIBUTES1 and SQL_DYNAMIC_CURSOR_ATTRIBUTES2 (to see if SQL_CA1_BULK_UPDATE_BY_BOOKMARK is returned, for instance).

After a call to SQLBulkOperations(), the block cursor position is undefined. The application has to call SQLFetchScroll() to set the cursor position. An application should only call SQLFetchScroll() with a *FetchOrientation* argument of SQL_FETCH_FIRST, SQL_FETCH_LAST, SQL_FETCH_ABSOLUTE, or SQL_FETCH_BOOKMARK. The cursor position is undefined if the application calls SQLFetch(), or SQLFetchScroll() with a *FetchOrientation* argument of SQL_FETCH_PRIOR, SQL_FETCH_NEXT, or SQL_FETCH_RELATIVE.

A column can be ignored in bulk operations (calls to SQLBulkOperations()). To do so, call SQLBindCol() and set the column length/indicator buffer (*StrLen_or_IndPtr*) to SQL_COLUMN_IGNORE. This does not apply to SQL_DELETE_BY_BOOKMARK bulk operation. See "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227 for more information.

It is not necessary for the application to set the SQL_ATTR_ROW_OPERATION_PTR statement attribute when calling SQLBulkOperations() because rows cannot be ignored when performing bulk operations with this function.

The buffer pointed to by the SQL_ATTR_ROWS_FETCHED_PTR statement attribute contains the number of rows affected by a call to SQLBulkOperations().

When the *Operation* argument is SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the select-list of the query specification associated with the cursor contains more than one reference to the same column, an error is generated.

Performing Bulk Inserts

To insert data with SQLBulkOperations(), an application performs the following sequence of steps:

1. Executes a query that returns a result set
2. Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that it wants to insert.
3. Calls SQLBindCol() to bind the data that it wants to insert. The data is bound to an array with a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE.

   Note: The size of the array pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute should either be equal to SQL_ATTR_ROW_ARRAY_SIZE or SQL_ATTR_ROW_STATUS_PTR should be a null pointer.

4. Calls SQLBulkOperations(*StatementHandle*, SQL_ADD) to perform the insertion.
5. If the application has set the SQL_ATTR_ROW_STATUS_PTR statement attribute, then it can inspect this array to see the result of the operation.

If an application binds column 0 before calling SQLBulkOperations() with an *Operation* argument of SQL_ADD, CLI will update the bound column 0 buffers with the bookmark values for the newly inserted row. For this to occur, the application must have set SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE before executing the statement.

Long data can be added in parts by SQLBulkOperations() using calls to SQLParamData() and SQLPutData(). For more information, see ″Providing Long Data for Bulk Inserts and Updates″ later in this section.

It is not necessary for the application to call SQLFetch() or SQLFetchScroll() before calling SQLBulkOperations().

If SQLBulkOperations() is called with an *Operation* argument of SQL_ADD on a cursor that contains duplicate columns, an error is returned.

Performing Bulk Updates Using Bookmarks

To perform bulk updates using bookmarks with SQLBulkOperations(), an application performs the following steps in sequence:

1. Sets the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE.
2. Executes a query that returns a result set.
3. Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that it wants to update.

   Calls SQLBindCol() to bind the data that it wants to update. The data is bound to an array with a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE. It also calls SQLBindCol() to bind column 0 (the bookmark column).
4. Copies the bookmarks for rows that it is interested in updating into the array bound to column 0.
5. Updates the data in the bound buffers.

   Note: The size of the array pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute should either be equal to SQL_ATTR_ROW_ARRAY_SIZE or SQL_ATTR_ROW_STATUS_PTR should be a null pointer.
6. Calls SQLBulkOperations()(*StatementHandle*, SQL_UPDATE_BY_BOOKMARK).

Note: If the application has set the SQL_ATTR_ROW_STATUS_PTR statement attribute, then it can inspect this array to see the result of the operation.

Optionally calls SQLBulkOperations(*StatementHandle*, SQL_FETCH_BY_BOOKMARK) to fetch data into the bound application buffers to verify that the update has occurred.

If data has been updated, CLI changes the value in the row status array for the appropriate rows to SQL_ROW_UPDATED.

Bulk updates performed by SQLBulkOperations() can include long data by using calls to SQLParamData() and SQLPutData(). For more information, see ″Providing Long Data for Bulk Inserts and Updates″ later in this section.

Bookmarks in DB2 CLI do not persist across cursors. This means that an application cannot use bookmarks that it has stored from a previous cursor. Instead, it has to call SQLFetch() or SQLFetchScroll() to retrieve the bookmarks before updating by bookmarks.

If SQLBulkOperations() is called with an *Operation* argument of SQL_UPDATE_BY_BOOKMARK on a cursor that contains duplicate columns, an error is returned.

Performing Bulk Fetches Using Bookmarks

To perform bulk fetches using bookmarks with SQLBulkOperations(), an application performs the following steps in sequence:

1. Sets the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE.
2. Executes a query that returns a result set.
3. Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that it wants to fetch.
4. Calls SQLBindCol() to bind the data that it wants to fetch. The data is bound to an array with a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE. It also calls SQLBindCol() to bind column 0 (the bookmark column).
5. Copies the bookmarks for rows that it is interested in fetching into the array bound to column 0. (This assumes that the application has already obtained the bookmarks separately.)

   Note: The size of the array pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute should either be equal to SQL_ATTR_ROW_ARRAY_SIZE, or the SQL_ATTR_ROW_STATUS_PTR statement attribute should be a null pointer.

6. Calls SQLBulkOperations(*StatementHandle*, SQL_FETCH_BY_BOOKMARK).
7. If the application has set the SQL_ATTR_ROW_STATUS_PTR statement attribute, then it can inspect this array to see the result of the operation.

Bookmarks in DB2 CLI do not persist across cursors. This means that an application cannot use bookmarks that it has stored from a previous cursor. Instead, it has to call SQLFetch() or SQLFetchScroll() to retrieve the bookmarks before updating by bookmarks.

Performing Bulk Deletes Using Bookmarks

To perform bulk deletes using bookmarks with SQLBulkOperations(), an application performs the following steps in sequence:

1. Sets the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE.
2. Executes a query that returns a result set.
3. Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that it wants to delete.
4. Calls SQLBindCol() to bind column 0 (the bookmark column).
5. Copies the bookmarks for rows that it is interested in deleting into the array bound to column 0.

   Note: The size of the array pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute should either be equal to SQL_ATTR_ROW_ARRAY_SIZE, or the SQL_ATTR_ROW_STATUS_PTR statement attribute should be a null pointer.
6. Calls SQLBulkOperations(*StatementHandle*, SQL_DELETE_BY_BOOKMARK).
7. If the application has set the SQL_ATTR_ROW_STATUS_PTR statement attribute, then it can inspect this array to see the result of the operation.

Bookmarks in DB2 CLI do not persist across cursors. This means that an application cannot use bookmarks that it has stored from a previous cursor. Instead, it has to call SQLFetch() or SQLFetchScroll() to retrieve the bookmarks before updating by bookmarks.

Providing Long Data for Bulk Inserts and Updates

Long data can be provided for bulk inserts and updates performed by calls to SQLBulkOperations(). To insert or update long data, an application performs the following steps in addition to the steps described in the ″Performing Bulk Inserts″ and ″Performing Bulk Updates Using Bookmarks″ sections earlier in this section.

1. When it binds the data using SQLBindCol(), the application places an application-defined value, such as the column number, in the **TargetValuePtr* buffer for data-at-execution columns. The value can be used later to identify the column.

   The application places the result of the SQL_LEN_DATA_AT_EXEC(*length*) macro in the **StrLen_or_IndPtr* buffer. If the SQL data type of the column is SQL_LONGVARBINARY, SQL_LONGVARCHAR, or a long, data source-specific data type and CLI returns ″Y″ for the SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo(), *length* is the number of bytes of data to be sent for the parameter; otherwise, it must be a non-negative value and is ignored.
2. When SQLBulkOperations() is called, if there are data-at-execution columns, the function returns SQL_NEED_DATA and proceeds to step 3 below. (If there are no data-at-execution columns, the process is complete.)
3. The application calls SQLParamData() to retrieve the address of the **TargetValuePtr* buffer for the first data-at-execution column to be processed. SQLParamData() returns SQL_NEED_DATA. The application retrieves the application-defined value from the **TargetValuePtr* buffer.

   Note: Although data-at-execution parameters are similar to data-at-execution columns, the value returned by SQLParamData() is different for each.

   Data-at-execution columns are columns in a rowset for which data will be sent with SQLPutData() when a row is updated or inserted with SQLBulkOperations(). They are bound with SQLBindCol(). The value returned by SQLParamData() is the address of the row in the **TargetValuePtr* buffer that is being processed.
4. The application calls SQLPutData() one or more times to send data for the column. More than one call is needed if all the data value cannot be returned in the **TargetValuePtr* buffer specified in SQLPutData(); note that multiple calls to SQLPutData() for the same column are allowed only when sending character C data to a column with a character, binary, or data source-specific data type or when sending binary C data to a column with a character, binary, or data source-specific data type.
5. The application calls SQLParamData() again to signal that all data has been sent for the column.
   - If there are more data-at-execution columns, SQLParamData() returns SQL_NEED_DATA and the address of the *TargetValuePtr* buffer for the next data-at-execution column to be processed. The application repeats steps 4 and 5 above.
   - If there are no more data-at-execution columns, the process is complete. If the statement was executed successfully, SQLParamData() returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO; if the execution failed,

it returns SQL_ERROR. At this point, SQLParamData() can return any SQLSTATE that can be returned by SQLBulkOperations().

If the operation is canceled, or an error occurs in SQLParamData() or SQLPutData(), after SQLBulkOperations() returns SQL_NEED_DATA, and before data is sent for all data-at-execution columns, the application can call only SQLCancel(), SQLGetDiagField(), SQLGetDiagRec(), SQLGetFunctions(), SQLParamData(), or SQLPutData() for the statement or the connection associated with the statement. If it calls any other function for the statement or the connection associated with the statement, the function returns SQL_ERROR and SQLSTATE HY010 (Function sequence error).

If the application calls SQLCancel() while CLI still needs data for data-at-execution columns, CLI cancels the operation. The application can then call SQLBulkOperations() again; canceling does not affect the cursor state or the current cursor position.

Row Status Array

The row status array contains status values for each row of data in the rowset after a call to SQLBulkOperations(). The status values in this array are set after a call to:

- SQLFetch()
- SQLFetchScroll()
- SQLSetPos()
- SQLBulkOperations()

This array is initially populated by a call to SQLBulkOperations() if SQLFetch() or SQLFetchScroll() has not been called prior to SQLBulkOperations(). This array is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute. The number of elements in the row status arrays must equal the number of rows in the rowset (as defined by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute). For information about this row status array, see “SQLFetch - Fetch Next Row” on page 396.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 28. SQLBulkOperations SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The *Operation* argument was SQL_FETCH_BY_BOOKMARK, and string or binary data returned for a column or columns with a data type of SQL_C_CHAR or SQL_C_BINARY resulted in the truncation of non-blank character or non-NULL binary data.
01S07	Invalid conversion.	The *Operation* argument was SQL_FETCH_BY_BOOKMARK, the data type of the application buffer was not SQL_C_CHAR or SQL_C_BINARY, and the data returned to application buffers for one or more columns was truncated. (For numeric C data types, the fractional part of the number was truncated. For time, timestamp, and interval C data types containing a time component, the fractional portion of the time was truncated.) (Function returns SQL_SUCCESS_WITH_INFO.)
07006	Restricted data type attribute violation.	The *Operation* argument was SQL_FETCH_BY_BOOKMARK, and the data value of a column in the result set could not be converted to the data type specified by the *TargetType* argument in the call to `SQLBindCol()`. The *Operation* argument was SQL_UPDATE_BY_BOOKMARK or SQL_ADD, and the data value in the application buffers could not be converted to the data type of a column in the result set.
07009	Invalid descriptor index.	The argument *Operation* was SQL_ADD and a column was bound with a column number greater than the number of columns in the result set.
21S02	Degree of derived table does not match column list.	The argument *Operation* was SQL_UPDATE_BY_BOOKMARK; and no columns were updatable because all columns were either unbound, read-only, or the value in the bound length/indicator buffer was SQL_COLUMN_IGNORE.
22001	String data right truncation.	The assignment of a character or binary value to a column in the result set resulted in the truncation of non-blank (for characters) or non-null (for binary) characters or bytes.

Table 28. SQLBulkOperations SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22003	Numeric value out of range.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of a numeric value to a column in the result set caused the whole (as opposed to fractional) part of the number to be truncated. The argument *Operation* was SQL_FETCH_BY_BOOKMARK, and returning the numeric value for one or more bound columns would have caused a loss of significant digits.
22007	Invalid datetime format.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of a date or timestamp value to a column in the result set caused the year, month, or day field to be out of range. The argument *Operation* was SQL_FETCH_BY_BOOKMARK, and returning the date or timestamp value for one or more bound columns would have caused the year, month, or day field to be out of range.
22008	Date/time field overflow.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the performance of datetime arithmetic on data being sent to a column in the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the natural rules for datetimes based on the Gregorian calendar. The *Operation* argument was SQL_FETCH_BY_BOOKMARK, and the performance of datetime arithmetic on data being retrieved from the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the natural rules for datetimes based on the Gregorian calendar.

Table 28. SQLBulkOperations SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22015	Interval field overflow.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of an exact numeric or interval C type to an interval SQL data type caused a loss of significant digits. The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK; when assigning to an interval SQL type, there was no representation of the value of the C type in the interval SQL type. The *Operation* argument was SQL_FETCH_BY_BOOKMARK, and assigning from an exact numeric or interval SQL type to an interval C type caused a loss of significant digits in the leading field. The *Operation* argument was SQL_FETCH_BY_BOOKMARK; when assigning to an interval C type, there was no representation of the value of the SQL type in the interval C type.
22018	Invalid character value for cast specification.	The *Operation* argument was SQL_FETCH_BY_BOOKMARK; the C type was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type. The argument *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK; the SQL type was an exact or approximate numeric, a datetime, or an interval data type; the C type was SQL_C_CHAR; and the value in the column was not a valid literal of the bound SQL type.
23000	Integrity constraint violation.	The *Operation* argument was SQL_ADD, SQL_DELETE_BY_BOOKMARK, or SQL_UPDATE_BY_BOOKMARK, and an integrity constraint was violated. The *Operation* argument was SQL_ADD, and a column that was not bound is defined as NOT NULL and has no default. The *Operation* argument was SQL_ADD, the length specified in the bound *StrLen_or_IndPtr* buffer was SQL_COLUMN_IGNORE, and the column did not have a default value.
24000	Invalid cursor state.	The *StatementHandle* was in an executed state but no result set was associated with the *StatementHandle*.

Table 28. SQLBulkOperations SQLSTATEs (continued)

SQLSTATE	Description	Explanation
40001	Serialization failure.	The transaction was rolled back due to a resource deadlock with another transaction.
40003	Statement completion unknown.	The associated connection failed during the execution of this function and the state of the transaction cannot be determined.
42000	Syntax error or access violation.	DB2 CLI was unable to lock the row as needed to perform the operation requested in the *Operation* argument.
44000	WITH CHECK OPTION violation.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the insert or update was performed on a viewed table or a table derived from the viewed table which was created by specifying WITH CHECK OPTION, such that one or more rows affected by the insert or update will no longer be present in the viewed table.
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation error.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY011	Operation invalid at this time.	The SQL_ATTR_ROW_STATUS_PTR statement attribute was set between calls to `SQLFetch()` or `SQLFetchScroll()` and `SQLBulkOperations`.

Table 28. SQLBulkOperations SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of this function.
HY090	Invalid string or buffer length.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, a data value was a null pointer, and the column length value was not 0, SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, a data value was not a null pointer; the C data type was SQL_C_BINARY or SQL_C_CHAR; and the column length value was less than 0, but not equal to SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NTS, or SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. The value in a length/indicator buffer was SQL_DATA_AT_EXEC; the SQL type was either SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long data type; and the SQL_NEED_LONG_DATA_LEN information type in `SQLGetInfo()` was "Y". The *Operation* argument was SQL_ADD, the SQL_ATTR_USE_BOOKMARK statement attribute was set to SQL_UB_VARIABLE, and column 0 was bound to a buffer whose length was not equal to the maximum length for the bookmark for this result set. (This length is available in the SQL_DESC_OCTET_LENGTH field of the IRD, and can be obtained by calling `SQLDescribeCol()`, `SQLColAttribute()`, or `SQLGetDescField()`.)

Table 28. SQLBulkOperations SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY092	Invalid attribute identifier.	The value specified for the *Operation* argument was invalid. The *Operation* argument was SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or SQL_DELETE_BY_BOOKMARK, and the SQL_ATTR_CONCURRENCY statement attribute was set to SQL_CONCUR_READ_ONLY. The *Operation* argument was SQL_DELETE_BY_BOOKMARK, SQL_FETCH_BY_BOOKMARK, or SQL_UPDATE_BY_BOOKMARK, and the bookmark column was not bound or the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.
HYC00	Optional feature not implemented.	DB2 CLI or data source does not support the operation requested in the *Operation* argument.
HYT00	Timeout expired.	The query timeout period expired before the data source returned the result set. The timeout period is set through `SQLSetStmtAttr()` with an *Attribute* argument of SQL_ATTR_QUERY_TIMEOUT.
HYT01	Connection timeout expired.	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through `SQLSetConnectAttr()`, SQL_ATTR_CONNECTION_TIMEOUT.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227
- "SQLCancel - Cancel Statement" on page 290
- "SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns" on page 408

- “SQLGetDescField - Get Single Field Settings of Descriptor Record” on page 458
- “SQLGetDescRec - Get Multiple Field Settings of Descriptor Record” on page 463
- “SQLSetDescField - Set a Single Field of a Descriptor Record” on page 649
- “SQLSetDescRec - Set Multiple Descriptor Fields for a Column or Parameter Data” on page 677
- “SQLSetPos - Set the Cursor Position in a Rowset” on page 691
- “SQLSetStmtAttr - Set Options Related to a Statement” on page 702

SQLCancel - Cancel Statement

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLCancel() can be used to prematurely terminate the *data-at-execution* sequence described in "Sending/Retrieving Long Data in Pieces" on page 80.

In a multi-threaded application, SQLCancel() will cancel the original request, which will return an SQLSTATE of **HY**008.

Syntax

```
SQLRETURN   SQLCancel       (SQLHSTMT         StatementHandle);  /* hstmt */
```

Function Arguments

Table 29. SQLCancel Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle

Usage

After SQLExecDirect() or SQLExecute() returns SQL_NEED_DATA to solicit for values for data-at-execution parameters, SQLCancel() can be used to cancel the data-at-execution sequence described in "Sending/Retrieving Long Data in Pieces" on page 80. SQLCancel() can be called any time before the final SQLParamData() in the sequence. After the cancellation of this sequence, the application can call SQLExecute() or SQLExecDirect() to re-initiate the data-at-execution sequence.

In DB2 CLI version 2, or when the SQL_ATTR_ODBC_VERSION environment attribute is set to SQL_OV_ODBC2, if an application calls SQLCancel() when no processing is being done on the statement, SQLCancel() has the same effect as SQLFreeStmt() with the SQL_CLOSE option. This is not the case in DB2 CLI version 5, or when the SQL_ATTR_ODBC_VERSION environment attribute is set to SQL_OV_ODBC5. A call to SQLCancel() when no processing is being done on the statement is not treated as SQLFreeStmt() with the SQL_CLOSE option, but has no effect at all. Applications should not call SQLCancel() to close a cursor, but rather SQLFreeStmt() should be used.

Canceling Asynchronous Processing

After an application calls a function asynchronously, it calls the function repeatedly to determine whether it has finished processing. If the function is

still processing, it returns SQL_STILL_EXECUTING. If the function has finished processing, it returns a different code.

After any call to the function that returns SQL_STILL_EXECUTING, an application can call SQLCancel() to cancel the function. If the cancel request is successful, SQL_SUCCESS is returned. This message does not indicate that the function was actually canceled; it indicates that the cancel request was processed. The application must continue to call the original function until the return code is not SQL_STILL_EXECUTING. If the function was successfully canceled, the return code is SQL_ERROR and SQLSTATE HY008 (Operation was cancelled). If the function completed its normal processing, the return code is SQL_SUCCESS or SQL_SUCCESS_WITH_INFO if the function succeeded or SQL_ERROR and a SQLSTATE other than HY008 (Operation was cancelled) if the function failed.

For more information about asynchronous processing, see "Asynchronous Execution of CLI" on page 138.

Canceling Functions in Multithread Applications

In a multithread application, the application can cancel a function that is running synchronously on a statement. To cancel the function, the application calls SQLCancel() with the same statement handle as that used by the target function, but on a different thread. How the function is canceled depends upon the operating system. As in canceling a function running asynchronously, the return code of the SQLCancel() indicates only whether DB2 CLI processed the request successfully. Only SQL_SUCCESS or SQL_ERROR can be returned; no SQLSTATEs are returned. If the original function is canceled, it returns SQL_ERROR and SQLSTATE HY008 (Operation was cancelled).

If an SQL statement is being executed when SQLCancel() is called on another thread to cancel the statement execution, it is possible that the execution succeeds and returns SQL_SUCCESS, while the cancel is also successful. In this case, DB2 CLI assumes that the cursor opened by the statement execution is closed by the cancel, so the application will not be able to use the cursor.

For more information about threading, see "Writing Multi-Threaded Applications" on page 46.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

Table 30. SQLCancel SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY018	Server declined cancel request.	The server declined the cancel request.
HY506	Error closing a file.	An error occurred when closing the temporary file generated by DB2 CLI when inserting LOB data in pieces using `SQLParamData()` / `SQLPutData()`.

Restrictions

None.

Example

Refer to “Example” on page 612.

References

- “SQLPutData - Passing Data Value for A Parameter” on page 609
- “SQLParamData - Get Next Parameter For Which A Data Value Is Needed” on page 575

SQLCloseCursor - Close Cursor and Discard Pending Results

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLCloseCursor() closes a cursor that has been opened on a statement, and discards pending results.

Syntax

```
SQLRETURN   SQLCloseCursor   (SQLHSTMT          StatementHandle);
```

Function Arguments

Table 31. SQLCloseCursor Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle

Usage

After an application calls SQLCloseCursor(), the application can reopen the cursor later by executing a **SELECT** statement again with the same or different parameter values.

SQLCloseCursor() returns SQLSTATE 24000 (Invalid cursor state) if no cursor is open. Calling SQLCloseCursor() is equivalent to calling SQLFreeStmt() with the SQL_CLOSE option, with the exception that SQLFreeStmt() with SQL_CLOSE has no effect on the application if no cursor is open on the statement, while SQLCloseCursor() returns SQLSTATE 24000 (Invalid cursor state).

Releasing Read Locks

The connection attribute SQL_ATTR_CLOSE_BEHAVIOR can be used to indicate whether or not DB2 CLI should attempt to release read locks acquired during a cursor's operation when the cursor is closed.

If SQL_ATTR_CLOSE_BEHAVIOR is set to SQL_CC_RELEASE then the database manager will attempt to release all read locks (if any) that have been held for the cursor. Read locks are IS, S, and U table locks as well as S, NS, and U row locks.

For more information on connection attributes see "SQLSetConnectAttr - Set Connection Attributes" on page 618, specifically "SQL_ATTR_CLOSE_BEHAVIOR Connection Attribute" on page 625. For more information on releasing read locks see:

- the “WITH RELEASE” clause of the “CLOSE statement” in the *SQL Reference*
- the “CLOSE CURSOR WITH RELEASE” section in the *Administration Guide*

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 32. SQLCloseCursor SQLSTATEs

SQLSTATE	Description	Explanation
01000	General warning	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state.	No cursor was open on the StatementHandle. (This is returned only by DB2 CLI Version 5 or later.)
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	An asynchronously executing function was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()` or `SQLExecDirect()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- “SQLCancel - Cancel Statement” on page 290
- “SQLFreeHandle - Free Handle Resources” on page 431
- “SQLMoreResults - Determine If There Are More Result Sets” on page 562

SQLColAttribute - Return a Column Attribute

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLColAttribute() returns descriptor information for a column in a result set. Descriptor information is returned as a character string, a 32-bit descriptor-dependent value, or an integer value.

Syntax

```
SQLRETURN   SQLColAttribute  (
                SQLHSTMT         StatementHandle,       /* hstmt */
                SQLSMALLINT      ColumnNumber,          /* icol */
                SQLSMALLINT      FieldIdentifier,       /* fDescType */
                SQLPOINTER       CharacterAttributePtr, /* rgbDesc */
                SQLSMALLINT      BufferLength,          /* cbDescMax */
                SQLSMALLINT      *StringLengthPtr,      /* pcbDesc */
                SQLPOINTER       NumericAttributePtr);  /* pfDesc */
```

Function Arguments

Table 33. SQLColAttribute Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLUSMALLINT	*ColumnNumber*	input	The number of the record in the IRD from which the field value is to be retrieved. This argument corresponds to the column number of result data, ordered sequentially from left to right, starting at 1. Columns may be described in any order. Column 0 can be specified in this argument, but all values except SQL_DESC_TYPE and SQL_DESC_OCTET_LENGTH will return undefined values.
SQLSMALLINT	*FieldIdentifier*	input	The field in row *ColumnNumber* of the IRD that is to be returned (see Table 34 on page 298).
SQLPOINTER	*CharacterAttributePtr*	output	Pointer to a buffer in which to return the value in the *FieldIdentifier* field of the *ColumnNumber* row of the IRD, if the field is a character string. Otherwise, the field is unused.
SQLINTEGER	BufferLength	input	The length of the **CharacterAttributePtr* buffer, if the field is a character string. Otherwise, this field is ignored.

Table 33. SQLColAttribute Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	*StringLengthPtr	output	Pointer to a buffer in which to return the total number of bytes (excluding the null termination byte for character data) available to return in **CharacterAttributePtr*. For character data, if the number of bytes available to return is greater than or equal to *BufferLength*, the descriptor information in **CharacterAttributePtr* is truncated to *BufferLength* minus the length of a null termination character and is null-terminated by DB2 CLI. For all other types of data, the value of *BufferLength* is ignored and DB2 CLI assumes the size of **CharacterAttributePtr* is 32 bits.
SQLPOINTER	*NumericAttributePtr*	output	Pointer to an integer buffer in which to return the value in the *FieldIdentifier* field of the *ColumnNumber* row of the IRD, if the field is a numeric descriptor type, such as SQL_DESC_COLUMN_LENGTH. Otherwise, the field is unused.

Usage

`SQLColAttribute()` returns information either in **NumericAttributePtr* or in **CharacterAttributePtr*. Integer information is returned in **NumericAttributePtr* as a 32-bit, signed value; all other formats of information are returned in **CharacterAttributePtr*. When information is returned in **NumericAttributePtr*, DB2 CLI ignores *CharacterAttributePtr*, *BufferLength*, and *StringLengthPtr* When information is returned in **CharacterAttributePtr*, DB2 CLI ignores *NumericAttributePtr*.

`SQLColAttribute()` returns values from the descriptor fields of the IRD. The function is called with a statement handle rather than a descriptor handle. The values returned by `SQLColAttribute()` for the *FieldIdentifier* values listed below can also be retrieved by calling `SQLGetDescField()` with the appropriate IRD handle.

The currently defined descriptor types, the version of DB2 CLI in which they were introduced (perhaps with different name), and the arguments in which information is returned for them are shown below; it is expected that more descriptor types will be defined to take advantage of different data sources.

DB2 CLI must return a value for each of the descriptor types. If a descriptor type does not apply to a data source, then, unless otherwise stated, DB2 CLI returns 0 in **StringLengthPtr* or an empty string in **CharacterAttributePtr*.

The following table lists the descriptor types returned by `SQLColAttribute()`.

Table 34. SQLColAttribute Arguments

FieldIdentifier	**Information returned in**	**Description**
SQL_COLUMN_AUTO_INCREMENT	Numeric AttributePtr	Changed to SQL_DESC_AUTO_UNIQUE_VALUE. [a]
SQL_COLUMN_CASE_SENSITIVE	Numeric AttributePtr	Changed to SQL_DESC_CASE_SENSITIVE. [a]
SQL_COLUMN_CATALOG_NAME	Character AttributePtr	Changed to SQL_DESC_CATALOG_NAME. [a]
SQL_COLUMN_COUNT	Numeric AttributePtr	Changed to SQL_DESC_COUNT. [a]
SQL_COLUMN_DISPLAY_SIZE	Numeric AttributePtr	Changed to SQL_DESC_DISPLAY_SIZE. [a]
SQL_COLUMN_LABEL	Character AttributePtr	Changed to SQL_DESC_LABEL. [a]
SQL_COLUMN_DISTINCT_TYPE	Character AttributePtr	Changed to SQL_DESC_DISTINCT_TYPE. [a]
SQL_COLUMN_LENGTH	Numeric AttributePtr	Changed to SQL_DESC_OCTET_LENGTH. [a]
SQL_COLUMN_MONEY	Numeric AttributePtr	Changed to SQL_DESC_FIXED_PREC_SCALE. [a]
SQL_COLUMN_NAME	Character AttributePtr	Changed to SQL_DESC_NAME. [a]
SQL_COLUMN_NULLABLE	Numeric AttributePtr	Changed to SQL_DESC_NULLABLE. [a]
SQL_COLUMN_OWNER_NAME	Character AttributePtr	Changed to SQL_DESC_SCHEMA_NAME. [a]
SQL_COLUMN_PRECISION	Numeric AttributePtr	Changed to SQL_DESC_PRECISION. [a]
SQL_COLUMN_QUALIFIER_NAME	Character AttributePtr	Changed to SQL_DESC_CATALOG_NAME. [a]
SQL_COLUMN_SCALE	Numeric AttributePtr	Changed to SQL_DESC_SCALE. [a]
SQL_COLUMN_SEARCHABLE	Numeric AttributePtr	Changed to SQL_DESC_SEARCHABLE. [a]

Table 34. SQLColAttribute Arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_COLUMN_TABLE_NAME	Character AttributePtr	Changed to SQL_DESC_TABLE_NAME. [a]
SQL_COLUMN_TYPE	Numeric AttributePtr	Changed to SQL_DESC_TYPE. [a]
SQL_COLUMN_TYPE_NAME	Character AttributePtr	Changed to SQL_DESC_TYPE_NAME. [a]
SQL_COLUMN_UNSIGNED	Numeric AttributePtr	Changed to SQL_DESC_UNSIGNED. [a]
SQL_COLUMN_UPDATABLE	Numeric AttributePtr	Changed to SQL_DESC_UPDATABLE. [a]
SQL_DESC_AUTO_UNIQUE_VALUE (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is an auto increment data type. SQL_FALSE is returned in *NumericAttributePtr* for all DB2 SQL data types.
SQL_DESC_BASE_COLUMN_NAME (DB2 CLI v5)	Character AttributePtr	The base column name for the set column. If a base column name does not exist (as in the case of columns that are expressions), then this variable contains an empty string. This information is returned from the SQL_DESC_BASE_COLUMN_NAME record field of the IRD, which is a read-only field.
SQL_DESC_BASE_TABLE_NAME (DB2 CLI v5)	Character AttributePtr	The name of the base table that contains the column. If the base table name cannot be defined or is not applicable, then this variable contains an empty string.
SQL_DESC_CASE_SENSITIVE (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is a case sensitive data type. Either SQL_TRUE or SQL_FALSE will be returned in *NumericAttributePtr* depending on the data type. Case sensitivity does not apply to graphic data types, SQL_FALSE is returned. SQL_FALSE is returned for non-character data types.

Table 34. SQLColAttribute Arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_CATALOG_NAME (DB2 CLI v2)	Character AttributePtr	The catalog of the table that contains the column is returned in *CharacterAttributePtr*. An empty string is returned since DB2 CLI only supports two part naming for a table.
SQL_DESC_CONCISE_TYPE (DB2 CLI v5)	Character AttributePtr	The concise data type. For the datetime data types, this field returns the concise data type, e.g., SQL_TYPE_TIME. This information is returned from the SQL_DESC_CONCISE_TYPE record field of the IRD.
SQL_DESC_COUNT (DB2 CLI v2)	Numeric AttributePtr	The number of columns in the result set is returned in *NumericAttributePtr*.
SQL_DESC_DISPLAY_SIZE (DB2 CLI v2)	Numeric AttributePtr	The maximum number of bytes needed to display the data in character form is returned in *NumericAttributePtr*. Refer to Table 197 on page 820 for the display size of each of the column types.
SQL_DESC_DISTINCT_TYPE (DB2 CLI v2)	Character AttributePtr	The user defined distinct type name of the column is returned in *CharacterAttributePtr*. If the column is a built-in SQL type and not a user defined distinct type, an empty string is returned. **Note:** This is an IBM defined extension to the list of descriptor attributes defined by ODBC.
SQL_DESC_FIXED_PREC_SCALE (DB2 CLI v2)	Numeric AttributePtr	SQL_TRUE if the column has a fixed precision and non-zero scale that are data-source-specific. SQL_FALSE if the column does not have a fixed precision and non-zero scale that are data-source-specific. SQL_FALSE is returned in *NumericAttributePtr* for all DB2 SQL data types.
SQL_DESC_LABEL (DB2 CLI v2)	Character AttributePtr	The column label is returned in *CharacterAttributePtr*. If the column does not have a label, the column name or the column expression is returned. If the column is unlabeled and unnamed, an empty string is returned.

Table 34. SQLColAttribute Arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_LENGTH (DB2 CLI v2)	Numeric AttributePtr	A numeric value that is either the maximum or actual character length of a character string or binary data type. It is the maximum character length for a fixed-length data type, or the actual character length for a variable-length data type. Its value always excludes the null termination byte that ends the character string. This information is returned from the SQL_DESC_LENGTH record field of the IRD.
SQL_DESC_LITERAL_PREFIX (DB2 CLI v5)	Character AttributePtr	This VARCHAR(128) record field contains the character or characters that DB2 CLI recognizes as a prefix for a literal of this data type. This field contains an empty string for a data type for which a literal prefix is not applicable.
SQL_DESC_LITERAL_SUFFIX (DB2 CLI v5)	Character AttributePtr	This VARCHAR(128) record field contains the character or characters that DB2 CLI recognizes as a suffix for a literal of this data type. This field contains an empty string for a data type for which a literal suffix is not applicable.
SQL_DESC_LOCAL_TYPE_NAME (DB2 CLI v5)	Character AttributePtr	This VARCHAR(128) record field contains any localized (native language) name for the data type that may be different from the regular name of the data type. If there is no localized name, then an empty string is returned. This field is for display purposes only. The character set of the string is locale-dependent and is typically the default character set of the server.

Table 34. SQLColAttribute Arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_NAME (DB2 CLI v2)	Character AttributePtr	The name of the column *ColumnNumber* is returned in *CharacterAttributePtr*. If the column is an expression, then the column number is returned. In either case, SQL_DESC_UNNAMED is set to SQL_NAMED. If there is no column name or a column alias, an empty string is returned and SQL_DESC_UNNAMED is set to SQL_UNNAMED. This information is returned from the SQL_DESC_NAME record field of the IRD. The column name value can be affected by the environment attribute SQL_ATTR_USE_LIGHT_OUTPUT_SQLDA. See “SQLSetEnvAttr - Set Environment Attribute” on page 682 for more information.
SQL_DESC_NULLABLE (DB2 CLI v2)	Numeric AttributePtr	If the column identified by *ColumnNumber* can contain nulls, then SQL_NULLABLE is returned in *NumericAttributePtr*. If the column is constrained not to accept nulls, then SQL_NO_NULLS is returned in *NumericAttributePtr*. This information is returned from the SQL_DESC_NULLABLE record field of the IRD.
SQL_DESC_NUM_PREX_RADIX (DB2 CLI v5)	Numeric AttributePtr	• If the datatype in the SQL_DESC_TYPE field is an approximate data type, this SQLINTEGER field contains a value of 2 because the SQL_DESC_PRECISION field contains the number of bits. • If the datatype in the SQL_DESC_TYPE field is an exact numeric data type, this field contains a value of 10 because the SQL_DESC_PRECISION field contains the number of decimal digits. • This field is set to 0 for all non-numeric data types.

Table 34. SQLColAttribute Arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_OCTET_LENGTH (DB2 CLI v2)	Numeric AttributePtr	The number of *bytes* of data associated with the column is returned in *NumericAttributePtr*. This is the length in bytes of data transferred on the fetch or `SQLGetData()` for this column if SQL_C_DEFAULT is specified as the C data type. Refer to Table 196 on page 819 for the length of each of the SQL data types. If the column identified in *ColumnNumber* is a fixed length character or binary string, (for example, SQL_CHAR or SQL_BINARY) the actual length is returned. If the column identified in *ColumnNumber* is a variable length character or binary string, (for example, SQL_VARCHAR or SQL_BLOB) the maximum length is returned.
SQL_DESC_PRECISION (DB2 CLI v2)	Numeric AttributePtr	The precision in units of digits is returned in *NumericAttributePtr* if the column is SQL_DECIMAL, SQL_NUMERIC, SQL_DOUBLE, SQL_FLOAT, SQL_INTEGER, SQL_REAL or SQL_SMALLINT. If the column is a character SQL data type, then the precision returned in *NumericAttributePtr*, indicates the maximum number of *characters* the column can hold. If the column is a graphic SQL data type, then the precision returned in *NumericAttributePtr*, indicates the maximum number of double-byte *characters* the column can hold. Refer to Table 194 on page 817 for the precision of each of the SQL data types. This information is returned from the SQL_DESC_PRECISION record field of the IRD.
SQL_DESC_SCALE (DB2 CLI v2)	Numeric AttributePtr	The scale attribute of the column is returned. Refer to Table 195 on page 819 for the scale of each of the SQL data types. This information is returned from the SCALE record field of the IRD.

Table 34. SQLColAttribute Arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_SCHEMA_NAME (DB2 CLI v2)	Character AttributePtr	The schema of the table that contains the column is returned in *CharacterAttributePtr*. An empty string is returned as DB2 CLI is unable to determine this attribute.
SQL_DESC_SEARCHABLE (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is searchable: • SQL_PRED_NONE (SQL_UNSEARCHABLE in DB2 CLI v2) if the column cannot be used in a WHERE clause. • SQL_PRED_CHAR (SQL_LIKE_ONLY in DB2 CLI v2) if the column can be used in a WHERE clause only with the LIKE predicate. • SQL_PRED_BASIC (SQL_ALL_EXCEPT_LIKE in DB2 CLI v2) if the column can be used in a WHERE clause with all comparison operators except LIKE. • SQL_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.
SQL_DESC_TABLE_NAME (DB2 CLI v2)	Character AttributePtr	The name of the table that contains the column is returned in *CharacterAttributePtr*. An empty string is returned as DB2 CLI cannot determine this attribute.
SQL_DESC_TYPE (DB2 CLI v2)	Numeric AttributePtr	The SQL data type of the column identified in *ColumnNumber* is returned in *NumericAttributePtr*. The possible values returned are listed in Table 2 on page 29. When *ColumnNumber* is equal to 0, SQL_BINARY is returned for variable-length bookmarks, and SQL_INTEGER is returned for fixed-length bookmarks. For the datetime data types, this field returns the verbose data type, i.e., SQL_DATETIME. This information is returned from the SQL_DESC_TYPE record field of the IRD.

Table 34. SQLColAttribute Arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_TYPE_NAME (DB2 CLI v2)	Character AttributePtr	The type of the column (as entered in an SQL statement) is returned in *CharacterAttributePtr*. For information on each data type refer to the TYPE_NAME attribute found in "Data Types and Data Conversion" on page 28.
SQL_DESC_UNNAMED (DB2 CLI v5)	Numeric AttributePtr	SQL_NAMED or SQL_UNNAMED. If the SQL_DESC_NAME field of the IRD contains a column alias, or a column name, SQL_NAMED is returned. If there is no column name or a column alias, SQL_UNNAMED is returned. This information is returned from the SQL_DESC_UNNAMED record field of the IRD.
SQL_DESC_UNSIGNED (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is an unsigned type or not. SQL_TRUE is returned in *NumericAttributePtr* for all non-numeric data types, SQL_FALSE is returned for all numeric data types.
SQL_DESC_UPDATABLE (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is an updateable data type: • SQL_ATTR_READWRITE_UNKNOWN is returned in *NumericAttributePtr* for all DB2 SQL data types. • SQL_ATTR_READONLY is returned if the column is obtained from a catalog function call. Although DB2 CLI does not retrun them, ODBC also defines the following values: • SQL_DESC_UPDATABLE • SQL_UPDT_WRITE

Note:

[a] In DB2 Version 5, the version 2 *FieldIdentifier* values have been changed. Although the old version 2 *FieldIdentifier* is still supported in this version of DB2, we recommend that you begin using the new *FieldIdentifier*.

This function is an extensible alternative to `SQLDescribeCol()`. `SQLDescribeCol()` returns a fixed set of descriptor information based on

SQLColAttribute

ANSI-89 SQL. `SQLColAttribute()` allows access to the more extensive set of descriptor information available in ANSI SQL-92 and DBMS vendor extensions.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 35. SQLColAttribute SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The buffer **CharacterAttributePtr* was not large enough to return the entire string value, so the string value was truncated. The length of the untruncated string value is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
07005	The statement did not return a result set.	The statement associated with the StatementHandle did not return a result set. There were no columns to describe.
07009	Invalid descriptor index.	The value specified for *ColumnNumber* was equal to 0, and the SQL_ATTR_USE_BOOKMARKS statement attribute was SQL_UB_OFF. The value specified for *ColumnNumber* was less than 0. The value specified for the argument *ColumnNumber* was greater than the number of columns in the result set.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.

Table 35. SQLColAttribute SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY008	Operation was cancelled	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, SQLCancel() was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, SQLCancel() was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The function was called prior to calling SQLPrepare() or SQLExecDirect() for the *StatementHandle*. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. SQLExecute() or SQLExecDirect() was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* was less than 0.
HY091	Descriptor type out of range.	The value specified for the argument *FieldIdentifier* was not one of the defined values, and was not an implementation-defined value.
HYC00	Driver not capable.	The value specified for the argument *FieldIdentifier* was not supported by DB2 CLI.

SQLColAttribute() can return any SQLSTATE that can be returned by SQLPrepare() or SQLExecute() when called after SQLPrepare() and before SQLExecute() depending on when the data source evaluates the SQL statement associated with the StatementHandle.

For performance reasons, an application should not call SQLColAttribute() before executing a statement.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227
- "SQLCancel - Cancel Statement" on page 290
- "SQLDescribeCol - Return a Set of Attributes for a Column" on page 336
- "SQLFetch - Fetch Next Row" on page 396
- "SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns" on page 408

SQLColAttributes - Get Column Attributes

Deprecated

Note:

In ODBC version 3, SQLColAttributes() has been deprecated and replaced with SQLColAttribute(); see "SQLColAttribute - Return a Column Attribute" on page 296 for more information.

Although this version of DB2 CLI continues to support SQLColAttributes(), we recommend that you begin using SQLColAttribute() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Migrating to the New Function

All of the field identifiers used with the version 2 function SQLColAttributes() have been changed for use with SQLColAttribute(). The old field identifiers are listed in Table 34 on page 298, along with their replacement values.

SQLSTATE 07002

All functions that return SQLSTATE **07**002 can also return the state if the application has used SQLSetColAttributes() to inform DB2 CLI of the descriptor information of the result set, but it did not provide this for every column in the result set. See "SQLState Cross Reference table" on page 799 for the list of DB2 CLI functions that return **07**002.

SQLColumnPrivileges - Get Privileges Associated With The Columns of A Table

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLColumnPrivileges() returns a list of columns and associated privileges for the specified table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated from a query.

Syntax

```
SQLRETURN SQLColumnPrivileges(
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLCHAR      *FAR CatalogName,       /* szCatalogName */
                SQLSMALLINT       NameLength1,       /* cbCatalogName */
                SQLCHAR      *FAR SchemaName,        /* szSchemaName */
                SQLSMALLINT       NameLength2,       /* cbSchemaName */
                SQLCHAR      *FAR TableName          /* szTableName */
                SQLSMALLINT       NameLength3,       /* cbTableName */
                SQLCHAR      *FAR ColumnName,        /* szColumnName */
                SQLSMALLINT       NameLength4);      /* cbColumnName */
```

Function Arguments

Table 36. SQLColumnPrivileges Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLCHAR *	CatalogName	input	Catalog qualifier of a 3 part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	NameLength1	input	Length of *CatalogName*. This must be set to 0.
SQLCHAR *	SchemaName	input	Schema qualifier of table name.
SQLSMALLINT	NameLength2	input	Length of *SchemaName*.
SQLCHAR *	TableName	input	Table name.
SQLSMALLINT	NameLength3	input	Length of *TableName*
SQLCHAR *	ColumnName	input	Buffer that may contain a *pattern-value* to qualify the result set by column name.
SQLSMALLINT	NameLength4	input	Length of *ColumnName*

Usage

The results are returned as a standard result set containing the columns listed in “Columns Returned by SQLColumnPrivileges” on page 311. The result set

is ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME, and PRIVILEGE. If multiple privileges are associated with any given column, each privilege is returned as a separate row. A typical application may wish to call this function after a call to SQLColumns() to determine column privilege information. The application should use the character strings returned in the TABLE_SCHEM, TABLE_NAME, COLUMN_NAME columns of the SQLColumns() result set as input arguments to this function.

Since calls to SQLColumnPrivileges() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating the calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Note that the *ColumnName* argument accepts a search pattern. For more information about valid search patterns, refer to “Input Arguments on Catalog Functions” on page 66.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns Returned by SQLColumnPrivileges

Column 1 TABLE_CAT (VARCHAR(128) Data type)
This is always NULL.

Column 2 TABLE_SCHEM (VARCHAR(128) Data type)
The name of the schema containing TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL Data type)
Name of the table or view.

Column 4 COLUMN_NAME (VARCHAR(128) not NULL Data type)
Name of the column of the specified table or view.

Column 5 GRANTOR (VARCHAR(128) Data type)
Authorization ID of the user who granted the privilege.

Column 6 GRANTEE (VARCHAR(128) Data type)
Authorization ID of the user to whom the privilege is granted.

Column 7 PRIVILEGE (VARCHAR(128) Data type)
The column privilege. This can be:

- INSERT
- REFERENCES
- SELECT
- UPDATE

Note: Some IBM RDBMSs do not offer column level privileges at the column level. DB2 Universal Database, DB2 for MVS/ESA and DB2 for VSE & VM support the UPDATE column privilege; there is one row in this result set for each updateable column. For all other privileges for DB2 Universal Database, DB2 for MVS/ESA and DB2 for VSE & VM, and for all privileges for other IBM RDBMSs, if a privilege has been granted at the table level, a row is present in this result set.

Column 8 IS_GRANTABLE (VARCHAR(3) Data type)
Indicates whether the grantee is permitted to grant the privilege to other users.

Either “YES” or “NO”.

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLColumnPrivileges()` result set in ODBC.

If there is more than one privilege associated with a column, then each privilege is returned as a separate row in the result set.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 37. SQLColumnPrivileges SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40001	Serialization failure	The transaction was rolled back due to a resource deadlock with another transaction.

Table 37. SQLColumnPrivileges SQLSTATEs (continued)

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error	An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()`, `SQLExecDirect()`, or `SQLSetPos()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY009	Invalid argument value.	*TableName* is NULL.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Restrictions

None.

Example

```
/* From CLI sample browser.c */
/* ... */
SQLRETURN list_column_privileges( SQLHANDLE hstmt,
                                  SQLCHAR * schema,
                                  SQLCHAR * tablename
                                ) {
/* ... */

    rc = SQLColumnPrivileges(hstmt, NULL, 0, schema, SQL_NTS,
                             tablename, SQL_NTS, columnname.s, SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) columnname.s, 129,
                    &columnname.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 5, SQL_C_CHAR, (SQLPOINTER) grantor.s, 129,
                    &grantor.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR, (SQLPOINTER) grantee.s, 129,
                    &grantee.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) privilege.s, 129,
                    &privilege.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 8, SQL_C_CHAR, (SQLPOINTER) is_grantable.s, 4,
                    &is_grantable.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf("Column Privileges for %s.%s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        sprintf((char *)cur_name, " Column: %s\n",  columnname.s);
        if (strcmp((char *)cur_name, (char *)pre_name) != 0) {
            printf("\n%s\n", cur_name);
            printf("   Grantor  Grantee   Privilege  Grantable\n");
            printf("   ---------------------------------------\n");
        }
        strcpy((char *)pre_name, (char *)cur_name);
        printf("   %-15s", grantor.s);
        printf(" %-15s", grantee.s);
        printf(" %-10s", privilege.s);
        printf(" %-3s\n", is_grantable.s);
    }                           /* endwhile */
```

References

- “SQLColumns - Get Column Information for a Table” on page 315
- “SQLTables - Get Table Information” on page 745

SQLColumns - Get Column Information for a Table

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLColumns() returns a list of columns in the specified tables. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to fetch a result set generated by a query.

Syntax

```
SQLRETURN   SQLColumns      (
                SQLHSTMT         StatementHandle,   /* hstmt */
                SQLCHAR      FAR *CatalogName,      /* szCatalogName */
                SQLSMALLINT      NameLength1,       /* cbCatalogName */
                SQLCHAR      FAR *SchemaName,       /* szSchemaName */
                SQLSMALLINT      NameLength2,       /* cbSchemaName */
                SQLCHAR      FAR *TableName,        /* szTableName */
                SQLSMALLINT      NameLength3,       /* cbTableName */
                SQLCHAR      FAR *ColumnName,       /* szColumnName */
                SQLSMALLINT      NameLength4);      /* cbColumnName */
```

Function Arguments

Table 38. SQLColumns Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLCHAR *	CatalogName	input	Buffer that may contain a *pattern-value* to qualify the result set. *Catalog* is the first part of a 3 part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	NameLength1	input	Length of *CatalogName*. This must be set to 0.
SQLCHAR *	SchemaName	input	Buffer that may contain a *pattern-value* to qualify the result set by schema name.
SQLSMALLINT	NameLength2	input	Length of *SchemaName*
SQLCHAR *	TableName	input	Buffer that may contain a *pattern-value* to qualify the result set by table name.
SQLSMALLINT	NameLength3	input	Length of *TableName*
SQLCHAR *	ColumnName	input	Buffer that may contain a *pattern-value* to qualify the result set by column name.
SQLSMALLINT	NameLength4	input	Length of *ColumnName*

Usage

This function is called to retrieve information about the columns of either a table or a set of tables. A typical application may wish to call this function after a call to SQLTables() to determine the columns of a table. The application should use the character strings returned in the TABLE_SCHEMA and TABLE_NAME columns of the SQLTables() result set as input to this function.

SQLColumns() returns a standard result set, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and ORDINAL_POSITION. “Columns Returned by SQLColumns()” on page 317 lists the columns in the result set.

The *SchemaName*, *TableName*, and *ColumnName* arguments accept search patterns. For more information about valid search patterns, see “Input Arguments on Catalog Functions” on page 66.

This function does not return information on the columns in a result set, SQLDescribeCol() or SQLColAttribute() should be used instead.

If the SQL_ATTR_LONGDATA_COMPAT attribute is set to SQL_LD_COMPAT_YES via either a call to SQLSetConnectAttr() or by setting the LONGDATACOMPAT keyword in the DB2 CLI initialization file, then the LOB data types are reported as SQL_LONGVARCHAR, SQL_LONGVARBINARY or SQL_LONGVARGRAPHIC.

Since calls to SQLColumns() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change. There were changes to these columns between version 2 and version 5. See “Changes to SQLColumns() Return Values” on page 771 for more information if you are running a version 2 DB2 CLI application (that uses SQLColumns()) against a version 5 or later server.

Optimize SQL Columns Keyword and Attribute

It is possible to set up the DB2 CLI/ODBC Driver to optimize calls to `SQLColumns()` using either:

- OPTIMIZESQLCOLUMNS DB2 CLI/ODBC configuration keyword (see "Configuring db2cli.ini" on page 159 for more information)
- SQL_ATTR_OPTIMIZESQLCOLUMNS connection attribute (see "SQLSetConnectAttr - Set Connection Attributes" on page 618 for more information)

If either of these values are set, then the following columns will not return any information:

- 12 REMARKS
- 13 COLUMN_DEF

Columns Returned by SQLColumns

Column 1 TABLE_CAT (VARCHAR(128))
This is always NULL.

Column 2 TABLE_SCHEM (VARCHAR(128))
The name of the schema containing TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL)
Name of the table, view, alias, or synonym.

Column 4 COLUMN_NAME (VARCHAR(128) not NULL)
Column identifier. Name of the column of the specified table, view, alias, or synonym.

Column 5 DATA_TYPE (SMALLINT not NULL)
SQL data type of column identified by COLUMN_NAME. This is one of the values in the Symbolic SQL Data Type column in Table 2 on page 29.

Column 6 TYPE_NAME (VARCHAR(128) not NULL)
Character string representing the name of the data type corresponding to DATA_TYPE.

Column 7 COLUMN_SIZE (INTEGER)
If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in characters for the column.

For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character.

For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.

See also, Table 194 on page 817.

Column 8 BUFFER_LENGTH (INTEGER)

The maximum number of bytes for the associated C buffer to store data from this column if SQL_C_DEFAULT were specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` calls. This length does not include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.

See also, Table 196 on page 819.

Column 9 DECIMAL_DIGITS (SMALLINT)

The scale of the column. NULL is returned for data types where scale is not applicable.

See also, Table 195 on page 819.

Column 10 NUM_PREC_RADIX (SMALLINT)

Either 10 or 2 or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, then the COLUMN_SIZE column contains the number of bits allowed in the column.

If DATA_TYPE is an exact numeric data type, this column contains the value 10 and the COLUMN_SIZE contains the number of decimal digits allowed for the column.

For numeric data types, the DBMS can return a NUM_PREC_RADIX of either 10 or 2.

NULL is returned for data types where radix is not applicable.

Column 11 NULLABLE (SMALLINT not NULL)

SQL_NO_NULLS if the column does not accept NULL values.

SQL_NULLABLE if the column accepts NULL values.

Column 12 REMARKS (VARCHAR(254))

May contain descriptive information about the column. It is possible that no information is returned in this column; see “Optimize SQL Columns Keyword and Attribute” on page 316 for more details.

Column 13 COLUMN_DEF (VARCHAR(254))

The column’s default value. If the default value is a numeric literal, then this column contains the character representation of the numeric literal with no enclosing single quotes. If the default value is a character string, then this column is that string enclosed in single quotes. If the default value a *pseudo-literal*, such as for DATE, TIME,

and TIMESTAMP columns, then this column contains the keyword of the pseudo-literal (e.g. CURRENT DATE) with no enclosing quotes.

If NULL was specified as the default value, then this column returns the word NULL, not enclosed in quotes. If the default value cannot be represented without truncation, then this column contains TRUNCATED with no enclosing single quotes. If no default value was specified, then this column is NULL.

It is possible that no information is returned in this column; see "Optimize SQL Columns Keyword and Attribute" on page 316 for more details.

Column 14 SQL_DATA_TYPE (SMALLINT not NULL)
SQL data type, as it appears in the SQL_DESC_TYPE record field in the IRD. This column is the same as the DATA_TYPE column.

Column 15 SQL_DATETIME_SUB (SMALLINT)
The subtype code for datetime data types:

- SQL_CODE_DATE
- SQL_CODE_TIME
- SQL_CODE_TIMESTAMP

For all other data types this column returns NULL.

Column 16 CHAR_OCTET_LENGTH (INTEGER)
Contains the maximum length in octets for a character data type column. For Single Byte character sets, this is the same as COLUMN_SIZE. For all other data types it is NULL.

Column 17 ORDINAL_POSITION (INTEGER not NULL)
The ordinal position of the column in the table. The first column in the table is number 1.

Column 18 IS_NULLABLE (VARCHAR(254))
Contains the string 'NO' if the column is known to be not nullable; and 'YES' otherwise.

Note: This result set is identical to the X/Open CLI `Columns()` result set specification, which is an extended version of the `SQLColumns()` result set specified in ODBC V2. The ODBC `SQLColumns()` result set includes every column in the same position.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR

- SQL_INVALID_HANDLE

Diagnostics

Table 39. SQLColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal SQL_NTS.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Restrictions

None.

Example

```
/* From CLI sample browser.c */
/* ... */
SQLRETURN list_columns( SQLHANDLE hstmt,
                        SQLCHAR * schema,
                        SQLCHAR * tablename
                      ) {
/* ... */

    rc = SQLColumns(hstmt, NULL, 0, schema, SQL_NTS,
                    tablename, SQL_NTS, (SQLCHAR *)"%", SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) column_name.s, 129,
                    &column_name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR, (SQLPOINTER) type_name.s, 129,
                    &type_name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 7, SQL_C_LONG, (SQLPOINTER) &length,
                    sizeof(length), &length_ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 9, SQL_C_SHORT, (SQLPOINTER) &scale,
                    sizeof(scale), &scale_ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 12, SQL_C_CHAR, (SQLPOINTER) remarks.s, 129,
                    &remarks.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 11, SQL_C_SHORT, (SQLPOINTER) & nullable,
                    sizeof(nullable), &nullable_ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf("Schema: %s  Table Name: %s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("   %s", column_name.s);
        if (nullable == SQL_NULLABLE) {
            printf(", NULLABLE");
        } else {
            printf(", NOT NULLABLE");
        }
        printf(", %s", type_name.s);
        if (length_ind != SQL_NULL_DATA) {
            printf(" (%ld", length);
        } else {
```

```
            printf("(\n");
        }
        if (scale_ind != SQL_NULL_DATA) {
            printf(", %d)\n", scale);
        } else {
            printf(")\n");
        }
    }                           /* endwhile */
```

References

- "SQLTables - Get Table Information" on page 745
- "SQLColumnPrivileges - Get Privileges Associated With The Columns of A Table" on page 310
- "SQLSpecialColumns - Get Special (Row Identifier) Columns" on page 726

SQLConnect - Connect to a Data Source

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLConnect() establishes a connection to the target database. The application must supply a target SQL database, and optionally an authorization-name, and an authentication-string.

A connection handle must be allocated using SQLAllocHandle() before calling this function.

This function must be called before allocating a statement handle using SQLAllocHandle().

Syntax

```
SQLRETURN   SQLConnect       (
                SQLHDBC           ConnectionHandle,  /* hdbc */
                SQLCHAR      *FAR ServerName,        /* szDSN */
                SQLSMALLINT       NameLength1,       /* cbDSN */
                SQLCHAR      *FAR UserName,          /* szUID */
                SQLSMALLINT       NameLength2,       /* cbUID */
                SQLCHAR      *FAR Authentication,    /* szAuthStr */
                SQLSMALLINT       NameLength3);      /* cbAuthStr */
```

Function Arguments

Table 40. SQLConnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle
SQLCHAR *	*ServerName*	input	Data Source: The name or alias-name of the database.
SQLSMALLINT	*NameLength1*	input	length of contents of *ServerName* argument
SQLCHAR *	*UserName*	input	Authorization-name (user identifier)
SQLSMALLINT	*NameLength2*	input	Length of contents of *UserName* argument
SQLCHAR *	*Authentication*	input	Authentication-string (password)
SQLSMALLINT	*NameLength3*	input	Length of contents of *Authentication* argument

Usage

The target database (also known as *data source*) for IBM RDBMSs is the database-alias. The application can obtain a list of databases available to connect to by calling SQLDataSources().

Before SQLDataSources() can return this information, the database(s) must be cataloged. Under Windows, using the ODBC Driver Manager, the user must catalog the database twice:

1. Once to the IBM RDBMS
2. Once to ODBC.

This can be accomplished in one step with the DB2 Client Setup included with the DB2 Client Application Enabler products. Although the methods of cataloging are different between ODBC Driver Manager and for IBM RDBMSs, the DB2 CLI applications are shielded from this. (One of the strengths of Call Level Interface is that the application does not have to know about the target database until SQLConnect() is invoked at runtime.) The mapping of the data source name to an actual DBMS is outside the scope and responsibility of the CLI application.

When using DB2 CLI in environments without an ODBC Driver Manager, the IBM RDBMSs need to be cataloged only once. For more information on cataloging, refer to "Chapter 4. Configuring CLI/ODBC and Running Sample Applications" on page 149.

The input length arguments to SQLConnect() (*NameLength1*, *NameLength2*, *NameLength3*) can be set to the actual length of their associated data (not including any null-terminating character) or to SQL_NTS to indicate that the associated data is null-terminated.

The *ServerName* and *UserName* argument values must not contain any blanks.

Use the more extensible SQLDriverConnect() function to connect when the applications needs to:

- Request the user to specify more than just the data source name, user ID, and password arguments on connect,
- Display a graphical dialog box to prompt for connect information

Various connection characteristics (options) may be specified by the end user in the section of the db2cli.ini (and odbc.ini) initialization file associated with the ServerName data source argument or set by the application using SQLSetConnectAttr(). The extended connect function, SQLDriverConnect(), can also be called with additional connect options.

Stored procedures written using DB2 CLI must make a *null* SQLConnect() call. A null SQLConnect() is where the *ServerName*, *UserName*, and *Authentication* argument pointers are all set to NULL and their respective length arguments all set to 0. A null SQLConnect() still requires SQLAllocEnv() and

SQLAllocConnect() be called first, but does not require that SQLTransact() be called before SQLDisconnect(). For more information, refer to "Stored Procedure Example" on page 131.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 41. SQLConnect SQLSTATEs

SQLSTATE	Description	Explanation
08001	Unable to connect to data source.	DB2 CLI was unable to establish a connection with the data source (server). The connection request was rejected because an existing connection established via embedded SQL already exists.
08002	Connection in use.	The specified *ConnectionHandle* has already been used to establish a connection with a data source and the connection is still open.
08004	The application server rejected establishment of the connection.	The data source (server) rejected the establishment of the connection. The number of connections specified by the MAXCONN keyword has been reached.
28000	Invalid authorization specification.	The value specified for the argument *UserName* or the value specified for the argument *Authentication* violated restrictions defined by the data source.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument *NameLength1* was less than 0, but not equal to SQL_NTS and the argument *ServerName* was not a null pointer. The value specified for argument *NameLength2* was less than 0, but not equal to SQL_NTS and the argument *UserName* was not a null pointer. The value specified for argument *NameLength3* was less than 0, but not equal to SQL_NTS and the argument *Authentication* was not a null pointer.

Table 41. SQLConnect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY501	Invalid data source name.	An invalid data source name was specified in argument *ServerName*.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

The implicit connection (or default database) option for IBM RDBMSs is not supported. SQLConnect() must be called before any SQL statements can be executed.

Example

```
/* From CLI sample samputil.c */
/* ... */
/*
 Global Variables for user id and password, defined in main module.
 To keep samples simple, not a recommended practice.
 The INIT_UID_PWD macro is used to initialize these variables.
*/
SQLCHAR server[SQL_MAX_DSN_LENGTH + 1] ;
SQLCHAR uid[MAX_UID_LENGTH + 1] ;
SQLCHAR pwd[MAX_PWD_LENGTH + 1] ;

/* ... */
/* connect without prompt */

SQLRETURN DBconnect( SQLHANDLE henv,
                     SQLHANDLE * hdbc
                   ) {

    /* allocate a connection handle */
    if ( SQLAllocHandle( SQL_HANDLE_DBC,
                         henv,
                         hdbc
                       ) != SQL_SUCCESS ) {
        printf( ">---ERROR while allocating a connection handle-----\n" ) ;
        return( SQL_ERROR ) ;
    }

    /* Set AUTOCOMMIT OFF */
    if ( SQLSetConnectAttr( * hdbc,
                            SQL_ATTR_AUTOCOMMIT,
```

```
                          ( void * ) SQL_AUTOCOMMIT_OFF, SQL_NTS
                        ) != SQL_SUCCESS ) {
        printf( ">---ERROR while setting AUTOCOMMIT OFF ------------\n" ) ;
        return( SQL_ERROR ) ;
    }

    if ( SQLConnect( * hdbc,
                     server, SQL_NTS,
                     uid,    SQL_NTS,
                     pwd,    SQL_NTS
                   ) != SQL_SUCCESS ) {
        printf( ">--- Error while connecting to database: %s -------\n",
                server
              ) ;
        SQLDisconnect( * hdbc ) ;
        SQLFreeHandle( SQL_HANDLE_DBC, * hdbc ) ;
        return( SQL_ERROR ) ;
    }
    else      /* Print Connection Information */
        printf( ">Connected to %s\n", server ) ;

    return( SQL_SUCCESS ) ;

}
```

References

- “SQLAllocHandle - Allocate Handle” on page 220
- “SQLDriverConnect - (Expanded) Connect to a Data Source” on page 350
- “SQLSetConnectAttr - Set Connection Attributes” on page 618
- “SQLGetConnectAttr - Get Current Attribute Setting” on page 439
- “SQLAllocHandle - Allocate Handle” on page 220
- “SQLDataSources - Get List of Data Sources” on page 332
- “SQLDisconnect - Disconnect from a Data Source” on page 347

SQLCopyDesc - Copy Descriptor Information Between Handles

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLCopyDesc() copies descriptor information from one descriptor handle to another.

Syntax

```
SQLRETURN   SQLCopyDesc      (SQLHDESC          SourceDescHandle,
                              SQLHDESC          TargetDescHandle);
```

Function Arguments

Table 42. SQLCopyDesc Arguments

Data Type	Argument	Use	Description
SQLHDESC	SourceDescHandle	input	Source descriptor handle.
SQLHDESC	TargetDescHandle	input	Target descriptor handle. *TargetDescHandle* can be a handle to an application descriptor or an IPD. SQLCopyDesc() will return SQLSTATE HY016 (Cannot modify an implementation descriptor) if *TargetDescHandle* is a handle to an IRD.

Usage

A call to SQLCopyDesc() copies the fields of the source descriptor handle to the target descriptor handle. Fields can only be copied to an application descriptor or an IPD, but not to an IRD. Fields can be copied from either an application or an implementation descriptor.

All fields of the descriptor, except SQL_DESC_ALLOC_TYPE (which specifies whether the descriptor handle was automatically or explicitly allocated), are copied, whether or not the field is defined for the destination descriptor. Copied fields overwrite the existing fields.

All descriptor fields are copied, even if *SourceDescHandle* and *TargetDescHandle* are are on two different connections or environments.

The call to SQLCopyDesc() is immediately aborted if an error occurs.

When the SQL_DESC_DATA_PTR field is copied, a consistency check is performed. If the consistency check fails, SQLSTATE HY021 (Inconsistent

descriptor information.) is returned and the call to SQLCopyDesc() is immediately aborted. For more information, see ”Consistency Checks“ in ”SQLSetDescField().“

Note: Descriptor handles can be copied across connections or environments. An application may, however, be able to associate an explicitly allocated descriptor handle with a *StatementHandle*, rather than calling SQLCopyDesc() to copy fields from one descriptor to another. An explicitly allocated descriptor can be associated with another *StatementHandle* on the same *ConnectionHandle* by setting the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC statement attribute to the handle of the explicitly allocated descriptor. When this is done, SQLCopyDesc() does not have to be called to copy descriptor field values from one descriptor to another.

A descriptor handle cannot be associated with a *StatementHandle* on another *ConnectionHandle*, however; to use the same descriptor field values on *StatementHandles* on different *ConnectionHandles*, SQLCopyDesc() has to be called.

For a description of the fields in a descriptor header or record, see “SQLSetDescField - Set a Single Field of a Descriptor Record” on page 649. For more information on descriptors, see “Using Descriptors” on page 97.

Copying Rows between Tables

An ARD on one statement handle can serve as the APD on another statement handle. This allows an application to copy rows between tables without copying data at the application level. To do this, an application calls SQLCopyDesc() to copy the fields of an ARD that describes a fetched row of a table, to the APD for a parameter in an INSERT statement on another statement handle. The SQL_ACTIVE_STATEMENTS *InfoType* returned by the driver for a call to SQLGetInfo() must be greater than 1 for this operation to succeed.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

When SQLCopyDesc() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec() with

a HandleType of SQL_HANDLE_DESC and a Handle of *TargetDescHandle*. If an invalid *SourceDescHandle* was passed in the call, SQL_INVALID_HANDLE will be returned, but no SQLSTATE will be returned.

When an error is returned, the call to SQLCopyDesc() is immediately aborted, and the contents of the fields in the *TargetDescHandle* descriptor are undefined.

Table 43. SQLCopyDesc SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was trying to connect failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by SQLGetDiagRec() in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate the memory required to support execution or completion of the function.
HY007	Associated statement is not prepared.	*SourceDescHandle* was associated with an IRD, and the associated statement handle was not in the prepared or executed state.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY016	Cannot modify an implementation row descriptor.	*TargetDescHandle* was associated with an IRD.
HY021	Inconsistent descriptor information.	The descriptor information checked during a consistency check was not consistent. For more information, see "Consistency Checks" on page 674 in SQLSetDescField().
HY092	Option type out of range.	The call to SQLCopyDesc() prompted a call to SQLSetDescField(), but **ValuePtr* was not valid for the *FieldIdentifier* argument on *TargetDescHandle*.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

None.

SQLDataSources - Get List of Data Sources

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLDataSources() returns a list of target databases available, one at a time. A database must be cataloged to be available. For more information on cataloging, refer to "Chapter 4. Configuring CLI/ODBC and Running Sample Applications" on page 149.

SQLDataSources() is usually called before a connection is made, to determine the databases that are available to connect to.

Syntax

```
SQLRETURN   SQLDataSources   (SQLHENV           EnvironmentHandle,
                              SQLUSMALLINT      Direction,
                              SQLCHAR      FAR  *ServerName,
                              SQLSMALLINT       BufferLength1,
                              SQLSMALLINT  FAR  *NameLength1Ptr,
                              SQLCHAR      FAR  *Description,
                              SQLSMALLINT       BufferLength2,
                              SQLSMALLINT  FAR  *NameLength2Ptr);
```

Function Arguments

Table 44. SQLDataSources Arguments

Data Type	Argument	Use	Description
SQLHENV	*EnvironmentHandle*	input	Environment handle.
SQLUSMALLINT	*Direction*	input	Used by application to request the first data source name in the list or the next one in the list. *Direction* can take on only the following values: • SQL_FETCH_FIRST • SQL_FETCH_NEXT
SQLCHAR *	*ServerName*	output	Pointer to buffer to hold the data source name retrieved.
SQLSMALLINT	*BufferLength1*	input	Maximum length of the buffer pointed to by *ServerName*. This should be less than or equal to SQL_MAX_DSN_LENGTH + 1.
SQLSMALLINT *	*NameLength1Ptr*	output	Pointer to location where the maximum number of bytes available to return in the *ServerName* will be stored.

Table 44. SQLDataSources Arguments (continued)

Data Type	Argument	Use	Description
SQLCHAR *	*Description*	output	Pointer to buffer where the description of the data source is returned. DB2 CLI will return the ***Comment*** field associated with the database catalogued to the DBMS.
SQLSMALLINT	*BufferLength2*	input	Maximum length of the *Description* buffer.
SQLSMALLINT *	*NameLength2Ptr*	output	Pointer to location where this function will return the actual number of bytes available to return for the description of the data source.

Usage

The application can call this function any time with *Direction* set to either SQL_FETCH_FIRST or SQL_FETCH_NEXT.

If SQL_FETCH_FIRST is specified, the first database in the list will always be returned.

If SQL_FETCH_NEXT is specified:

- Directly following a SQL_FETCH_FIRST call, the second database in the list is returned
- Before any other `SQLDataSources()` call, the first database in the list is returned
- When there are no more databases in the list, SQL_NO_DATA_FOUND is returned. If the function is called again, the first database is returned.
- Any other time, the next database in the list is returned.

In an ODBC environment, the ODBC Driver Manager will perform this function. For more information refer to “Appendix C. DB2 CLI and ODBC” on page 779.

Since the IBM RDBMSs always returns the description of the data source blank padded to 30 bytes, DB2 CLI will do the same.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQLDataSources

Diagnostics

Table 45. SQLDataSources SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data source name returned in the argument *ServerName* was longer than the value specified in the argument *BufferLength1*. The argument *NameLength1Ptr* contains the length of the full data source name. (Function returns SQL_SUCCESS_WITH_INFO.) The data source name returned in the argument *Description* was longer than the value specified in the argument *BufferLength2*. The argument *NameLength2Ptr* contains the length of the full data source description. (Function returns SQL_SUCCESS_WITH_INFO.)
58004	Unexpected system failure.	Unrecoverable system error.
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no specific SQLSTATE was defined. The error message returned by `SQLError()` in the argument *ErrorMsg* describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument *BufferLength1* was less than 0. The value specified for argument *BufferLength2* was less than 0.
HY103	Direction option out of range.	The value specified for the argument *Direction* was not equal to SQL_FETCH_FIRST or SQL_FETCH_NEXT.

Authorization

None.

Example

```
/* From CLI sample datasour.c */
/* ... */

#include <stdio.h>
#include <stdlib.h>
#include <sqlcli1.h>
#include "samputil.h"          /* Header file for CLI sample code */

/* ... */

/************************************************************************
```

```
** main
** - initialize
** - terminate
*****************************************************************/
int main() {

    SQLHANDLE henv ;
    SQLRETURN rc ;
    SQLCHAR source[SQL_MAX_DSN_LENGTH + 1], description[255] ;
    SQLSMALLINT buffl, desl ;

/* ... */

    /* allocate an environment handle */
    rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    /* list the available data sources (servers) */
    printf( "The following data sources are available:\n" ) ;
    printf( "ALIAS NAME                      Comment(Description)\n" ) ;
    printf( "-------------------------------------------------\n" ) ;

    while ( ( rc = SQLDataSources( henv,
                                   SQL_FETCH_NEXT,
                                   source,
                                   SQL_MAX_DSN_LENGTH + 1,
                                   &buffl,
                                   description,
                                   255,
                                   &desl
                                 )
            ) != SQL_NO_DATA_FOUND
          ) printf( "%-30s  %s\n", source, description ) ;

    rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    return( SQL_SUCCESS ) ;


}
```

References

None.

SQLDescribeCol - Return a Set of Attributes for a Column

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLDescribeCol() returns a set of commonly used descriptor information (column name, type, precision, scale, nullability) for the indicated column in the result set generated by a query.

This information is also available in the fields of the IRD.

If the application needs only one attribute of the descriptor information, or needs an attribute not returned by SQLDescribeCol(), the SQLColAttribute() function can be used in place of SQLDescribeCol(). See "SQLColAttribute - Return a Column Attribute" on page 296 for more information.

Either SQLPrepare() or SQLExecDirect() must be called before calling this function.

This function (or SQLColAttribute()) is usually called before a bind column function (SQLBindCol(), SQLBindFileToCol()) to determine the attributes of a column before binding it to an application variable.

Syntax

```
SQLRETURN   SQLDescribeCol   (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLUSMALLINT      ColumnNumber,      /* icol */
                SQLCHAR      *FAR ColumnName,        /* szColName */
                SQLSMALLINT       BufferLength,      /* cbColNameMax */
                SQLSMALLINT  *FAR NameLengthPtr,     /* pcbColName */
                SQLSMALLINT  *FAR DataTypePtr,       /* pfSqlType */
                SQLUINTEGER  *FAR ColumnSizePtr,     /* pcbColDef */
                SQLSMALLINT  *FAR DecimalDigitsPtr,  /* pibScale */
                SQLSMALLINT  *FAR NullablePtr);      /* pfNullable */
```

Function Arguments

Table 46. SQLDescribeCol Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLUSMALLINT	*ColumnNumber*	input	Column number to be described. Columns are numbered sequentially from left to right, starting at 1. This can also be set to 0 to describe the bookmark column.

Table 46. SQLDescribeCol Arguments (continued)

Data Type	Argument	Use	Description
SQLCHAR *	*ColumnName*	output	Pointer to column name buffer. This value is read from the SQL_DESC_NAME field of the IRD. This is set to NULL if the column name cannot be determined. The column name value can be affected by the environment attribute SQL_ATTR_USE_LIGHT_OUTPUT_SQLDA. See "SQLSetEnvAttr - Set Environment Attribute" on page 682 for more information.
SQLSMALLINT	*BufferLength*	input	Size of *ColumnName* buffer.
SQLSMALLINT *	*NameLengthPtr*	output	Bytes available to return for *ColumnName* argument. Truncation of column name (*ColumnName*) to *BufferLength - 1* bytes occurs if *NameLengthPtr* is greater than or equal to *BufferLength*.
SQLSMALLINT *	*DataTypePtr*	output	Base SQL data type of column. To determine if there is a User Defined Type associated with the column, call `SQLColAttribute()` with *fDescType* set to SQL_COLUMN_DISTINCT_TYPE. Refer to the Symbolic SQL Data Type column of Table 2 on page 29 for the data types that are supported.
SQLUINTEGER *	*ColumnSizePtr*	output	Precision of column as defined in the database. If *fSqlType* denotes a graphic or DBCLOB SQL data type, then this variable indicates the maximum number of double-byte *characters* the column can hold.
SQLSMALLINT *	*DecimalDigitsPtr*	output	Scale of column as defined in the database (only applies to SQL_DECIMAL, SQL_NUMERIC, SQL_TIMESTAMP). Refer to Table 195 on page 819 for the scale of each of the SQL data types.
SQLSMALLINT *	*NullablePtr*	output	Indicates whether NULLS are allowed for this column • SQL_NO_NULLS • SQL_NULLABLE

Usage

Columns are identified by a number, are numbered sequentially from left to right, and may be described in any order.

- Column numbers start at 1 if bookmarks are not used (SQL_ATTR_USE_BOOKMARKS statement attribute set to SQL_UB_OFF).
- Column numbers start at 0 if bookmarks are used (the statement attribute set to SQL_UB_ON).

If a null pointer is specified for any of the pointer arguments, DB2 CLI assumes that the information is not needed by the application and nothing is returned.

If the column is a User Defined Type, SQLDescribeCol only returns the built-in type in *DataTypePtr*. Call SQLColAttribute() with *fDescType* set to SQL_COLUMN_DISTINCT_TYPE to obtain the User Defined Type.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

If SQLDescribeCol() returns either SQL_ERROR, or SQL_SUCCESS_WITH_INFO, one of the following SQLSTATEs may be obtained by calling the SQLError() function.

Table 47. SQLDescribeCol SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The column name returned in the argument *ColumnName* was longer than the value specified in the argument *BufferLength*. The argument *NameLengthPtr* contains the length of the full column name. (Function returns SQL_SUCCESS_WITH_INFO.)
07005	The statement did not return a result set.	The statement associated with the *StatementHandle* did not return a result set. There were no columns to describe. (Call SQLNumResultCols() first to determine if there are any rows in the result set.)
07009	Invalid descriptor index	The value specified for *ColumnNumber* was equal to 0, and the SQL_ATTR_USE_BOOKMARKS statement attribute was SQL_UB_OFF. The value specified for *ColumnNumber* was less than 0. The value specified for the argument *ColumnNumber* was greater than the number of columns in the result set.

Table 47. SQLDescribeCol SQLSTATEs (continued)

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY002	Invalid column number.	The value specified for the argument *ColumnNumber* was less than 1. The value specified for the argument *ColumnNumber* was greater than the number of columns in the result set.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY090	Invalid string or buffer length.	The length specified in argument *BufferLength* less than 1.
HY010	Function sequence error.	The function was called prior to calling `SQLPrepare()` or `SQLExecDirect()` for the *StatementHandle*. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HYC00	Driver not capable.	The SQL data type of column *ColumnNumber* is not recognized by DB2 CLI.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Restrictions

The following ODBC defined data types are not supported:

- SQL_BIT
- SQL_TINYINT

Example

```
/* From CLI sample samputil.c */
/* ... */
/* print_results */

SQLRETURN print_results( SQLHANDLE hstmt ) {

    SQLCHAR     colname[32] ;
    SQLSMALLINT coltype ;
    SQLSMALLINT colnamelen ;
    SQLSMALLINT nullable ;
    SQLUINTEGER collen[MAXCOLS] ;
    SQLSMALLINT scale ;
    SQLINTEGER  outlen[MAXCOLS] ;
    SQLCHAR *   data[MAXCOLS] ;
    SQLCHAR     errmsg[256] ;
    SQLRETURN   rc ;
    SQLSMALLINT nresultcols, i ;
    SQLINTEGER  displaysize ;

    rc = SQLNumResultCols( hstmt, &nresultcols ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
    for ( i = 0; i < nresultcols; i++ ) {
        SQLDescribeCol( hstmt,
                        ( SQLSMALLINT ) ( i + 1 ),
                        colname,
                        sizeof(colname),
                        &colnamelen,
                        &coltype,
                        &collen[i],
                        &scale,
                        NULL
                      ) ;
        /* get display length for column */
        SQLColAttribute( hstmt,
                         ( SQLSMALLINT ) ( i + 1 ),
                         SQL_DESC_DISPLAY_SIZE,
                         NULL,
                         0,
                         NULL,
                         &displaysize
                       ) ;

        /*
         Set column length to max of display length,
         and column name length. Plus one byte for
         null terminator.
```

```
    */
    collen[i] = max( displaysize,
                     strlen( ( char * ) colname )
                   ) + 1 ;

    printf( "%-*.*s",
            ( int ) collen[i],
            ( int ) collen[i],
            colname
          ) ;

    /* allocate memory to bind column */
    data[i] = ( SQLCHAR * ) malloc( ( int ) collen[i] ) ;

    /* bind columns to program vars, converting all types to CHAR */
    SQLBindCol( hstmt,
                ( SQLSMALLINT ) ( i + 1 ),
                SQL_C_CHAR,
                data[i],
                collen[i],
                &outlen[i]
              ) ;
}

printf( "\n" ) ;
/* display result rows */
while ( SQLFetch( hstmt ) != SQL_NO_DATA ) {
    errmsg[0] = '\0' ;
    for ( i = 0; i < nresultcols; i++ ) {
        /* Check for NULL data */
        if ( outlen[i] == SQL_NULL_DATA )
           printf( "%-*.*s",
                   ( int ) collen[i],
                   ( int ) collen[i],
                   "NULL"
                 ) ;
        else { /* Build a truncation message for any columns truncated */
           if ( outlen[i] >= collen[i] ) {
              sprintf( ( char * ) errmsg + strlen( ( char * ) errmsg ),
                       "%d chars truncated, col %d\n",
                       ( int ) outlen[i] - collen[i] + 1,
                       i + 1
                     ) ;
           }
           /* Print column */
           printf( "%-*.*s",
                   ( int ) collen[i],
                   ( int ) collen[i],
                   data[i]
                 ) ;
        }
    }                          /* for all columns in this row  */

    printf( "\n%s", errmsg ) ; /* print any truncation messages */
}                              /* while rows to fetch */
```

```
    /* free data buffers */
    for ( i = 0; i < nresultcols; i++ ) {
        free( data[i] ) ;
    }

    return( SQL_SUCCESS ) ;

}                               /* end print_results */
```

References

- “SQLSetColAttributes - Set Column Attributes” on page 617
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLNumResultCols - Get Number of Result Columns” on page 572
- “SQLPrepare - Prepare a Statement” on page 583

SQLDescribeParam - Return Description of a Parameter Marker

Purpose

Specification:	DB2 CLI 5.0	ODBC 1.0	ISO CLI

SQLDescribeParam() returns the description of a parameter marker associated with a prepared SQL statement. This information is also available in the fields of the IPD.

Syntax

```
SQLRETURN   SQLDescribeParam (SQLHSTMT          StatementHandle,
                              SQLUSMALLINT      ParameterNumber,
                              SQLSMALLINT       *DataTypePtr,
                              SQLUINTEGER       *ParameterSizePtr,
                              SQLSMALLINT       *DecimalDigitsPtr,
                              SQLSMALLINT       *NullablePtr);
```

Function Arguments

Table 48. SQLDescribeParam Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLUSMALLINT	ParameterNumber	input	Parameter marker number ordered sequentially in increasing parameter order, starting at 1.
SQLSMALLINT *	DataTypePtr	output	Pointer to a buffer in which to return the SQL data type of the parameter. This value is read from the SQL_DESC_CONCISE_TYPE record field of the IPD. When ColumnNumber is equal to 0 (for a bookmark column), SQL_BINARY is returned in **DataTypePtr* for variable-length bookmarks.
SQLUINTEGER *	ParameterSizePtr	output	Pointer to a buffer in which to return the size of the column or expression of the corresponding parameter marker as defined by the data source.
SQLSMALLINT	DecimalDigitsPtr	output	Pointer to a buffer in which to return the number of decimal digits of the column or expression of the corresponding parameter as defined by the data source.

Table 48. SQLDescribeParam Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	NullablePtr	output	Pointer to a buffer in which to return a value that indicates whether the parameter allows NULL values. This value is read from the SQL_DESC_NULLABLE field of the IPD. One of the following: • SQL_NO_NULLS: The parameter does not allow NULL values (this is the default value). • SQL_NULLABLE: The parameter allows NULL values. • SQL_NULLABLE_UNKNOWN: Cannot determine if the parameter allows NULL values.

Usage

Parameter markers are numbered in increasing parameter order, starting with 1, in the order they appear in the SQL statement.

`SQLDescribeParam()` does not return the type (input, input/output, or output) of a parameter in an SQL statement. Except in calls to procedures, all parameters in SQL statements are input parameters. To determine the type of each parameter in a call to a procedure, an application calls `SQLProcedureColumns()`.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 49. SQLDescribeParam SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)

Table 49. SQLDescribeParam SQLSTATEs (continued)

SQLSTATE	Description	Explanation
07009	Invalid descriptor index.	The value specified for the argument *ParameterNumber* less than 1. The value specified for the argument *ParameterNumber* was greater than the number of parameters in the associated SQL statement. The parameter marker was part of a non-DML statement. The parameter marker was part of a SELECT list.
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was connected failed before the function completed processing.
21S01	Insert value list does not match column list.	The number of parameters in the INSERT statement did not match the number of columns in the table named in the statement.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY008	Operation was cancelled	Asynchronous processing was enabled for the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The function was called prior to calling `SQLPrepare()` or `SQLExecDirect()` for the *StatementHandle*. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()` `SQLExecDirect()`, `SQLBulkOperations()`, or `SQLSetPos()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

Table 49. SQLDescribeParam SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY013	Unexpected memory handling error.	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator" on page 247
- "SQLCancel - Cancel Statement" on page 290
- "SQLExecute - Execute a Statement" on page 373
- "SQLPrepare - Prepare a Statement" on page 583

SQLDisconnect - Disconnect from a Data Source

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLDisconnect() closes the connection associated with the database connection handle.

SQLEndTran() must be called before calling SQLDisconnect() if an outstanding transaction exists on this connection.

After calling this function, either call SQLConnect() to connect to another database, or call SQLFreeHandle().

Syntax

```
SQLRETURN   SQLDisconnect    (SQLHDBC          ConnectionHandle;) /* hdbc */
```

Function Arguments

Table 50. SQLDisconnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle

Usage

If an application calls SQLDisconnect() before it has freed all the statement handles associated with the connection, DB2 CLI frees them after it successfully disconnects from the database.

If SQL_SUCCESS_WITH_INFO is returned, it implies that even though the disconnect from the database is successful, additional error or implementation specific information is available. For example, a problem was encountered on the clean up subsequent to the disconnect, or if there is no current connection because of an event that occurred independently of the application (such as communication failure).

After a successful SQLDisconnect() call, the application can re-use *ConnectionHandle* to make another SQLConnect() or SQLDriverConnect() request.

An application should not rely on SQLDisconnect() to close cursors (with both stored procedures and regular client applications). In both cases the cursor should be closed using SQLCloseCursor(), then the statement handle freed with a call to SQLFreeHandle() with a *HandleType* of SQL_HANDLE_STMT.

SQLDisconnect

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 51. SQLDisconnect SQLSTATEs

SQLSTATE	Description	Explanation
01002	Disconnect error.	An error occurred during the disconnect. However, the disconnect succeeded. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The connection specified in the argument *ConnectionHandle* was not open.
25000 **25**501	Invalid transaction state.	There was a transaction in process on the connection specified by the argument *ConnectionHandle*. The transaction remains active, and the connection cannot be disconnected. **Note:** This error does not apply to stored procedures written in DB2 CLI.
25501	Invalid transaction state.	There was a transaction in process on the connection specified by the argument *ConnectionHandle*. The transaction remains active, and the connection cannot be disconnected.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “SQLAllocHandle - Allocate Handle” on page 220

References

- “SQLAllocHandle - Allocate Handle” on page 220
- “SQLConnect - Connect to a Data Source” on page 323
- “SQLDriverConnect - (Expanded) Connect to a Data Source” on page 350
- “SQLEndTran - End Transactions of a Connection” on page 356
- “SQLFreeHandle - Free Handle Resources” on page 431

SQLDriverConnect - (Expanded) Connect to a Data Source

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLDriverConnect() is an alternative to SQLConnect(). Both functions establish a connection to the target database, but SQLDriverConnect() supports additional connection parameters and the ability to prompt the user for connection information.

Use SQLDriverConnect() when the data source requires parameters other than the 3 input arguments supported by SQLConnect() (data source name, user ID and password), or when you want to use DB2 CLI's graphical user interface to prompt the user for mandatory connection information.

Once a connection is established, the completed connection string is returned. Applications can store this string for future connection requests.

Syntax

Generic

```
SQLRETURN   SQLDriverConnect (
                SQLHDBC           ConnectionHandle,  /* hdbc */
                SQLHWND           WindowHandle,      /* hwnd */
                SQLCHAR      *FAR InConnectionString, /* szConnStrIn */
                SQLSMALLINT       StringLength1,     /* cbConnStrIn */
                SQLCHAR      *FAR OutConnectionString,/* szConnStrOut */
                SQLSMALLINT       BufferLength,      /* cbConnStrOutMax */
                SQLSMALLINT  *FAR StringLength2Ptr,   /* pcbConnStrOut */
                SQLUSMALLINT      DriverCompletion); /* fDriverCompletion */
```

Function Arguments

Table 52. SQLDriverConnect Arguments

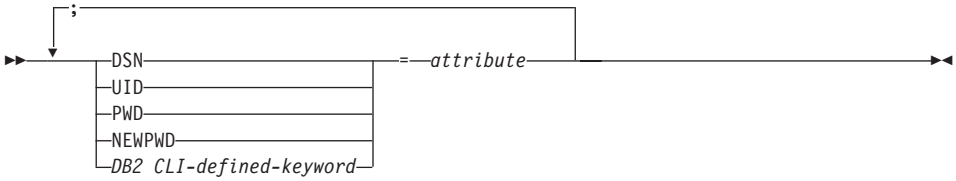
Data Type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle
SQLHWND	*hwindow*	input	Window handle (platform dependent): on Windows, this is the parent Windows handle. On OS/2, this is the parent PM window handle. On AIX, this is the parent MOTIF Widget window handle. If a NULL is passed, then no dialog will be presented.

Table 52. SQLDriverConnect Arguments (continued)

Data Type	Argument	Use	Description
SQLCHAR *	*InConnectionString*	input	A full, partial or empty (null pointer) connection string (see syntax and description below).
SQLSMALLINT	*StringLength1*	input	Length of *InConnectionString*.
SQLCHAR *	*OutConnectionString*	output	Pointer to buffer for the completed connection string. If the connection was established successfully, this buffer will contain the completed connection string. Applications should allocate at least SQL_MAX_OPTION_STRING_LENGTH bytes for this buffer.
SQLSMALLINT	*BufferLength*	input	Maximum size of the buffer pointed to by *OutConnectionString*.
SQLCHAR *	*StringLength2Ptr*	output	Pointer to the number of bytes available to return in the *OutConnectionString* buffer. If the value of *StringLength2Ptr* is greater than or equal to *BufferLength*, the completed connection string in *OutConnectionString* is truncated to *BufferLength* - 1 bytes.
SQLUSMALLINT	*DriverCompletion*	input	Indicates when DB2 CLI should prompt the user for more information. Possible values: • SQL_DRIVER_PROMPT • SQL_DRIVER_COMPLETE • SQL_DRIVER_COMPLETE_REQUIRED • SQL_DRIVER_NOPROMPT

Usage

The connection string is used to pass one or more values needed to complete a connection. The contents of the connection string and the value of *DriverCompletion* will determine if DB2 CLI needs to establish a dialog with the user.



Each keyword above has an attribute that is equal to the following:

DSN Data source name. The name or alias-name of the database. Required if *DriverCompletion* is equal to SQL_DRIVER_NOPROMPT.

UID Authorization-name (user identifier).

PWD The password corresponding to the authorization name. If there is no password for the user ID, an empty value is specified (PWD=;).

NEWPWD
New password used as part of a change password request. The application can either specify the new string to use, for example, NEWPWD=anewpass; or specify NEWPWD=; and rely on a dialog box generated by the DB2 CLI driver to prompt for the new password (set the *DriverCompletion* argument to anything other than SQL_DRIVER_NOPROMPT).

The list of DB2 CLI defined keywords and their associated attribute values are discussed in “Configuration Keywords” on page 164. Any one of the keywords in that section can be specified on the connection string. If any keywords are repeated in the connection string, the value associated with the first occurrence of the keyword is used.

If any keywords exists in the CLI initialization file, the keywords and their respective values are used to augment the information passed to DB2 CLI in the connection string. If the information in the CLI initialization file contradicts information in the connection string, the values in connection string take precedence.

If the end user *Cancel*s a dialog box presented, SQL_NO_DATA_FOUND is returned.

The following values of *DriverCompletion* determines when a dialog will be opened:

SQL_DRIVER_PROMPT:
A dialog is always initiated. The information from the connection string and the CLI initialization file are used as initial values, to be supplemented by data input via the dialog box.

SQL_DRIVER_COMPLETE:
A dialog is only initiated if there is insufficient information in the connection string. The information from the connection string is used as initial values, to be supplemented by data entered via the dialog box.

SQL_DRIVER_COMPLETE_REQUIRED:
A dialog is only initiated if there is insufficient information in the connection string. The information from the connection string is used as initial values. Only mandatory information is requested. The user is prompted for required information only.

SQL_DRIVER_NOPROMPT:
The user is not prompted for any information. A connection is attempted with the information contained in the connection string. If there is not enough information, SQL_ERROR is returned.

Once a connection is established, the complete connection string is returned. Applications that need to set up multiple connections to the same database for a given user ID should store this output connection string. This string can then be used as the input connection string value on future `SQLDriverConnect()` calls.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

All of the diagnostics generated by “SQLConnect - Connect to a Data Source” on page 323 can be returned here as well. The following table shows the additional diagnostics that can be returned.

Table 53. SQLDriverConnect SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer *szConnstrOut* was not large enough to hold the entire connection string. The argument *StringLength2Ptr* contains the actual length of the connection string available for return. (Function returns SQL_SUCCESS_WITH_INFO)

Table 53. SQLDriverConnect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
01S00	Invalid connection string attribute.	An invalid keyword or attribute value was specified in the input connection string, but the connection to the data source was successful anyway because one of the following occurred: • The unrecognized keyword was ignored. • The invalid attribute value was ignored, the default value was used instead. (Function returns SQL_SUCCESS_WITH_INFO)
HY000	General error. Dialog Failed	The information specified in the connection string was insufficient for making a connect request, but the dialog was prohibited by setting *fCompletion* to SQL_DRIVER_NOPROMPT. The attempt to display the dialog failed.
HY090	Invalid string or buffer length.	The value specified for *StringLength1* was less than 0, but not equal to SQL_NTS. The value specified for *BufferLength* was less than 0.
HY110	Invalid driver completion.	The value specified for the argument *fCompletion* was not equal to one of the valid values.

Restrictions

None.

Example

```
/* From CLI sample drivrcon.c */
/* ... */
/*********************************************************************
**   drv_connect - Prompt for connect options and connect              **
*********************************************************************/

int
drv_connect(SQLHENV henv,
            SQLHDBC * hdbc,
            SQLCHAR con_type)
{
    SQLRETURN       rc;
    SQLCHAR         server[SQL_MAX_DSN_LENGTH + 1];
    SQLCHAR         uid[MAX_UID_LENGTH + 1];
    SQLCHAR         pwd[MAX_PWD_LENGTH + 1];
    SQLCHAR         con_str[255];
    SQLCHAR         buffer[255];
    SQLSMALLINT     outlen;

    printf("Enter Server Name:\n");
```

```
    gets((char *) server);
    printf("Enter User Name:\n");
    gets((char *) uid);
    printf("Enter Password Name:\n");
    gets((char *) pwd);

    /* Allocate a connection handle */
    SQLAllocHandle( SQL_HANDLE_DBC,
                        henv,
                        hdbc
                      );
    CHECK_HANDLE( SQL_HANDLE_DBC, *hdbc, rc);

    sprintf((char *)con_str, "DSN=%s;UID=%s;PWD=%s;AUTOCOMMIT=0;
            CONNECTTYPE=1;", server, uid, pwd);

    rc = SQLDriverConnect(*hdbc,
            (SQLHWND) NULL,
            con_str,
            SQL_NTS,
            NULL, 0, NULL,
            SQL_DRIVER_NOPROMPT);
    if (rc != SQL_SUCCESS) {
        printf("Error while connecting to database, RC= %ld\n", rc);
        CHECK_HANDLE( SQL_NULL_HENV, *hdbc, rc);
        return (SQL_ERROR);
    } else {
        printf("Successful Connect\n");
        return (SQL_SUCCESS);
    }
}
```

References

- “SQLAllocHandle - Allocate Handle” on page 220
- “SQLConnect - Connect to a Data Source” on page 323

SQLEndTran - End Transactions of a Connection

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLEndTran() requests a commit or rollback operation for all active operations on all statements associated with a connection. SQLEndTran() can also request that a commit or rollback operation be performed for all connections associated with an environment.

Syntax

```
SQLRETURN   SQLEndTran      (SQLSMALLINT      HandleType,
                             SQLHANDLE        Handle,
                             SQLSMALLINT      CompletionType);
```

Function Arguments

Table 54. SQLEndTran Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	*Handle* type identifier. Contains either SQL_HANDLE_ENV if *Handle* is an environment handle, or SQL_HANDLE_DBC if *Handle* is a connection handle.
SQLHANDLE	Handle	input	The handle, of the type indicated by *HandleType*, indicating the scope of the transaction. See the "Usage" section below for more information.
SQLSMALLINT	*CompletionType*	input	One of the following two values: • SQL_COMMIT • SQL_ROLLBACK

Usage

If *HandleType* is SQL_HANDLE_ENV and *Handle* is a valid environment handle, then DB2 CLI will attempt to commit or roll back transactions one at a time, depending on the value of *CompletionType*, on all connections that are in a connected state on that environment. SQL_SUCCESS will only be returned if it receives SQL_SUCCESS for each connection. If it receives SQL_ERROR on one or more connections, it will return SQL_ERROR to the application, and the diagnostic information will be placed in the diagnostic data structure of the environment. To determine which connection(s) failed during the commit or rollback operation, the application can call SQLGetDiagRec() for each connection.

SQLEndTran() should not be used when working in a Distributed Unit of Work environment. The transaction manager APIs should be used instead.

If *CompletionType* is SQL_COMMIT, SQLEndTran() issues a commit request for all active operations on any statement associated with an affected connection. If *CompletionType* is SQL_ROLLBACK, SQLEndTran() issues a rollback request for all active operations on any statement associated with an affected connection. If no transactions are active, SQLEndTran() returns SQL_SUCCESS with no effect on any data sources.

If DB2 CLI is in manual-commit mode (by calling SQLSetConnectAttr() with the SQL_ATTR_AUTOCOMMIT attribute set to SQL_AUTOCOMMIT_OFF), a new transaction is implicitly started when an SQL statement that can be contained within a transaction is executed against the current data source.

To determine how transaction operations affect cursors, an application calls SQLGetInfo() with the SQL_CURSOR_ROLLBACK_BEHAVIOR and SQL_CURSOR_COMMIT_BEHAVIOR options.

If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value equals SQL_CB_DELETE, SQLEndTran() closes and deletes all open cursors on all statements associated with the connection and discards all pending results. SQLEndTran() leaves any statement present in an allocated (unprepared) state; the application can reuse them for subsequent SQL requests or can call SQLFreeStmt() or SQLFreeHandle() with a *HandleType* of SQL_HANDLE_STMT to deallocate them.

If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value equals SQL_CB_CLOSE, SQLEndTran() closes all open cursors on all statements associated with the connection. SQLEndTran() leaves any statement present in a prepared state; the application can call SQLExecute() for a statement associated with the connection without first calling SQLPrepare().

If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value equals SQL_CB_PRESERVE, SQLEndTran() does not affect open cursors associated with the connection. Cursors remain at the row they pointed to prior to the call to SQLEndTran().

When autocommit mode is on, calling SQLEndTran() with either SQL_COMMIT or SQL_ROLLBACK when no transaction is active will return SQL_SUCCESS (indicating that there is no work to be committed or rolled back) and have no effect on the data source.

SQLEndTran

When autocommit mode is off, calling SQLEndTran() with a *CompletionType* of either SQL_COMMIT or SQL_ROLLBACK always returns SQL_SUCCESS.

When a DB2 CLI application is running in autocommit mode, the DB2 CLI driver does not pass the SQLEndTran() statement to the server.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 55. SQLEndTran SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The *ConnectionHandle* was not in a connected state.
08007	Connection failure during transaction.	The connection associated with the *ConnectionHandle* failed during the execution of the function and it cannot be determined whether the requested **COMMIT** or **ROLLBACK** occurred before the failure.
40001	Transaction rollback.	The transaction was rolled back due to a resource deadlock with another transaction.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by SQLGetDiagRec() in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	An asynchronously executing function was called for a *StatementHandle* associated with the *ConnectionHandle* and was still executing when SQLEndTran() was called. SQLExecute() or SQLExecDirect() was called for a *StatementHandle* associated with the *ConnectionHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY012	Invalid transaction code.	The value specified for the argument *CompletionType* was neither SQL_COMMIT nor SQL_ROLLBACK.
HY092	Option type out of range.	The value specified for the argument *HandleType* was neither SQL_HANDLE_ENV nor SQL_HANDLE_DBC.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLGetInfo - Get General Information" on page 489
- "SQLFreeHandle - Free Handle Resources" on page 431
- "SQLFreeStmt - Free (or Reset) a Statement Handle" on page 435

SQLError - Retrieve Error Information

Deprecated

Note:

In ODBC version 3, SQLError() has been deprecated and replaced with SQLGetDiagRec() and SQLGetDiagField(); see "SQLGetDiagRec - Get Multiple Fields Settings of Diagnostic Record" on page 477 and "SQLGetDiagField - Get a Field of Diagnostic Data" on page 468 for more information.

Although this version of DB2 CLI continues to support SQLError(), we recommend that you begin using SQLGetDiagRec() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Purpose

Specification:	**DB2 CLI** 1.1	**ODBC** 1.0	**ISO CLI**

SQLError() returns the diagnostic information (both errors and warnings) associated with the most recently invoked DB2 CLI function for a particular statement, connection or environment handle.

The information consists of a standardized SQLSTATE, native error code, and a text message. Refer to "Diagnostics" on page 25 for more information.

Call SQLError() after receiving a return code of SQL_ERROR or SQL_SUCCESS_WITH_INFO from another function call.

Note: Some database servers may provide product-specific diagnostic information after returning SQL_NO_DATA_FOUND from the execution of a statement.

Syntax

```
SQLRETURN   SQLError         (SQLHENV           henv,
                              SQLHDBC           hdbc,
                              SQLHSTMT          hstmt,
                              SQLCHAR      FAR  *szSqlState,
                              SQLINTEGER   FAR  *pfNativeError,
                              SQLCHAR      FAR  *szErrorMsg,
                              SQLSMALLINT       cbErrorMsgMax,
                              SQLSMALLINT  FAR  *pcbErrorMsg);
```

Function Arguments

Table 56. SQLError Arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle. To obtain diagnostic information associated with an environment, pass a valid environment handle. Set *hdbc* and *hstmt* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
SQLHDBC	*hdbc*	input	Database connection handle. To obtain diagnostic information associated with a connection, pass a valid database connection handle, and set *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored.
SQLHSTMT	*hstmt*	input	Statement handle. To obtain diagnostic information associated with a statement, pass a valid statement handle. The *henv* and *hdbc* arguments are ignored.
SQLCHAR *	*szSqlState*	output	SQLSTATE as a string of 5 characters terminated by a null character. The first 2 characters indicate error class; the next 3 indicate subclass. The values correspond directly to SQLSTATE values defined in the X/Open SQL CAE specification and the ODBC specification, augmented with IBM specific and product specific SQLSTATE values.
SQLINTEGER *	*pfNativeError*	output	Native error code. In DB2 CLI, the *pfNativeError* argument will contain the SQLCODE value returned by the DBMS. If the error is generated by DB2 CLI and not the DBMS, then this field will be set to -99999.

Table 56. SQLError Arguments (continued)

Data Type	Argument	Use	Description

SQLCHAR *	*szErrorMsg*	output	Pointer to buffer to contain the implementation defined message text. If the error is detected by DB2 CLI, then the error message would be prefaced by:
			
			`[IBM][CLI Driver]`
			
			to indicate that it is DB2 CLI that detected the error and there is no database connection yet.
			
			If the error is detected while there is a database connection, then the error message returned from the DBMS would be prefaced by:
			
			`[IBM][CLI Driver][`*`DBMS-name`*`]`
			
			where `DBMS-name` is the name returned by `SQLGetInfo()` with SQL_DBMS_NAME information type.
			
			For example,
			• `DB2`
			• `DB2/6000`
			
			If the DBMS-name is not regcognized then DB2 CLI will treat it as a DB2 Universal Database version 5 data source.
			
			If the error is generated by the DBMS, the IBM defined SQLSTATE will be appended to the text string.

SQLSMALLINT	*cbErrorMsgMax*	input	The maximum (that is, the allocated) length of the buffer *szErrorMsg*. The recommended length to allocate is SQL_MAX_MESSAGE_LENGTH + 1.
SQLSMALLINT *	*pcbErrorMsg*	output	Pointer to total number of bytes available to return to the *szErrorMsg* buffer. This does not include the null termination character.

Usage

The SQLSTATEs are those defined by the X/OPEN SQL CAE and the X/Open SQL CLI CAE, augmented with IBM specific and product specific SQLSTATE values.

To obtain diagnostic information associated with:

- An environment, pass a valid environment handle. Set *hdbc* and *hstmt* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
- A connection, pass a valid database connection handle, and set *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored.
- A statement, pass a valid statement handle. The *henv* and *hdbc* arguments are ignored.

If diagnostic information generated by one DB2 CLI function is not retrieved before a function other than `SQLError()` is called with the same handle, the information for the previous function call is lost. This is true whether or not diagnostic information is generated for the second DB2 CLI function call.

Multiple diagnostic messages may be available after a given DB2 CLI function call. These messages can be retrieved one at a time by repeatedly calling `SQLError()`. For each message retrieved, `SQLError()` returns SQL_SUCCESS and removes it from the list of messages available. When there are no more messages to retrieve, SQL_NO_DATA_FOUND is returned, the SQLSTATE is set to "00000", *pfNativeError* is set to 0, and *pcbErrorMsg* and *szErrorMsg* are undefined.

Diagnostic information stored under a given handle is cleared when a call is made to `SQLError()` with that handle, or when another DB2 CLI function call is made with that handle. However, information associated with a given handle type is not cleared by a call to `SQLError()` with an associated but different handle type: for example, a call to `SQLError()` with a connection handle input will not clear errors associated with any statement handles under that connection.

SQL_SUCCESS is returned even if the buffer for the error message (*szErrorMsg*) is too short since the application will not be able to retrieve the same error message by calling `SQLError()` again. The actual length of the message text is returned in the *pcbErrorMsg*.

To avoid truncation of the error message, declare a buffer length of SQL_MAX_MESSAGE_LENGTH + 1. The message text will never be longer than this.

Note: The defined value of SQL_MAX_MESSAGE_LENGTH has been increased since DB2 CLI Version 1.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND is returned if no diagnostic information is available for the input handle, or if all of the messages have been retrieved via calls to `SQLError()`.

Diagnostics

SQLSTATEs are not defined, since `SQLError()` does not generate diagnostic information for itself.

Restrictions

Although ODBC also returns X/Open SQL CAE SQLSTATEs, only DB2 CLI (and the DB2 ODBC driver) returns the additional IBM defined SQLSTATEs. The ODBC Driver Manager also returns SQLSTATE values with a prefix of **IM**. These SQLSTATES are not defined by X/Open and are not returned by DB2 CLI. For more information on ODBC specific SQLSTATEs refer to *ODBC 3.0 Software Development Kit and Programmer's Reference.*

Because of this, you should only build dependencies on the standard SQLSTATEs. This means any branching logic in the application should only rely on the standard SQLSTATEs. The augmented SQLSTATEs are most useful for debugging purposes.

Note: It may be useful to build dependencies on the class (the first 2 characters) of the SQLSTATEs.

Example

Refer to "SQLGetDiagRec - Get Multiple Fields Settings of Diagnostic Record" on page 477.

References

- "SQLGetSQLCA - Get SQLCA Data Structure" on page 541

SQLExecDirect - Execute a Statement Directly

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLExecDirect() directly executes the specified SQL statement. The statement can only be executed once. Also, the connected database server must be able to dynamically prepare statement. (For more information about supported SQL statements refer to Table 215 on page 843.)

Syntax

```
SQLRETURN   SQLExecDirect    (SQLHSTMT          StatementHandle,
                              SQLCHAR      *FAR StatementText,
                              SQLINTEGER         TextLength);
```

Function Arguments

Table 57. SQLExecDirect Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle. There must not be an open cursor associated with *StatementHandle*, see "SQLFreeStmt - Free (or Reset) a Statement Handle" on page 435 for more information.
SQLCHAR *	*StatementText*	input	SQL statement string. The connected database server must be able to prepare the statement, see Table 215 on page 843 for more information.
SQLINTEGER	*TextLength*	input	Length of contents of *StatementText* argument. The length must be set to either the exact length of the statement, or if the statement is null-terminated, set to SQL_NTS.

Usage

If the SQL statement text contains vendor escape clause sequences, DB2 CLI will first modify the SQL statement text to the appropriate DB2 specific format before submitting it for preparation and execution. If the application does not generate SQL statements that contain vendor escape clause sequences ("Using Vendor Escape Clauses" on page 144), then it should set the SQL_ATTR_NOSCAN statement attribute to SQL_NOSCAN_ON at the connection level so that DB2 CLI does not perform a scan for vendor escape clauses.

The SQL statement cannot be a COMMIT or ROLLBACK. Instead, SQLTransact() must be called to issue COMMIT or ROLLBACK. For more information about supported SQL statements refer to Table 215 on page 843.

The SQL statement string may contain parameter markers. A parameter marker is represented by a "?" character, and is used to indicate a position in the statement where an application supplied value is to be substituted when SQLExecDirect() is called. This value can be obtained from:

- An application variable.

  SQLSetParam() or SQLBindParameter() is used to bind the application storage area to the parameter marker.
- A LOB value residing at the server referenced by a LOB locator. SQLBindParameter() or SQLSetParam() is used to bind a LOB locator to a parameter marker. The actual value of the LOB is kept at the server and does not need to be first transferred to the application before being used as input parameter value for another SQL statement.
- A file (within the applications environment) containing a LOB value.

  SQLBindFileToParam() is used to bind a file to a LOB parameter marker. When SQLExecDirect() is executed, DB2 CLI will transfer the contents of the file directly to the database server.

All parameters must be bound before calling SQLExecDirect().

Refer to the PREPARE section of the *SQL Reference* for information on rules related to parameter markers.

If the SQL statement is a query, SQLExecDirect() will generate a cursor name, and open the cursor. If the application has used SQLSetCursorName() to associate a cursor name with the statement handle, DB2 CLI associates the application generated cursor name with the internally generated one.

If a result set is generated, SQLFetch() or SQLFetchScroll() will retrieve the next row (or rows) of data into bound variables, LOB locators or LOB file references (using SQLBindCol() or SQLBindFileToCol()). Data can also be retrieved by calling SQLGetData() for any column that was not bound.

If the SQL statement is a Positioned DELETE or a Positioned UPDATE, the cursor referenced by the statement must be positioned on a row and must be defined on a separate statement handle under the same connection handle.

There must not already be an open cursor on the statement handle.

If SQLParamOptions() has been called to specify that an array of input parameter values has been bound to each parameter marker, then the application needs to call SQLExecDirect() only once to process the entire array of input parameter values.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA
- SQL_NO_DATA_FOUND

SQL_NEED_DATA is returned when the application has requested to input data-at-execution parameter values by calling SQLParamData() and SQLPutData().

SQL_NO_DATA_FOUND is returned if the SQL statement is a Searched UPDATE or Searched DELETE and no rows satisfy the search condition.

Diagnostics

Table 58. SQLExecDirect SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	*StatementText* contained an UPDATE or DELETE statement which did not contain a WHERE clause. (Function returns SQL_SUCCESS_WITH_INFO or SQL_NO_DATA_FOUND if there were no rows in the table).
01508	Statement disqualified for blocking.	The statement was disqualified for blocking for reasons other than storage.
07001	Wrong number of parameters.	The number of parameters bound to application variables using SQLBindParameter() was less than the number of parameter markers in the SQL statement contained in the argument *StatementText*.
07006	Invalid conversion.	Transfer of data between DB2 CLI and the application variables would result in incompatible data conversion.
21S01	Insert value list does not match column list.	*StatementText* contained an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*StatementText* contained a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.

Table 58. SQLExecDirect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22001	String data right truncation.	A character string assigned to a character type column exceeded the maximum length of the column.
22003	Numeric value out of range.	A numeric value assigned to a numeric type column caused truncation of the whole part of the number, either at the time of assignment or in computing an intermediate result. *StatementText* contained an SQL statement with an arithmetic expression which caused division by zero. **Note:** as a result the cursor state is undefined for DB2 Universal Database (the cursor will remain open for other RDBMSs).
22005	Error in assignment.	*StatementText* contained an SQL statement with a parameter or literal and the value or LOB locator was incompatible with the data type of the associated table column. The length associated with a parameter value (the contents of the *pcbValue* buffer specified on `SQLBindParameter()`) is not valid. The argument *fSQLType* used in `SQLBindParameter()` or `SQLSetParam()`, denoted an SQL graphic data type, but the deferred length argument (*pcbValue*) contains an odd length value. The length value must be even for graphic data types.
22007	Invalid datetime format.	*StatementText* contained an SQL statement with an invalid datetime format; that is, an invalid string representation or value was specified, or the value was an invalid date.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	*StatementText* contained an SQL statement with an arithmetic expression that caused division by zero.
23000	Integrity constraint violation.	The execution of the SQL statement is not permitted because the execution would cause integrity constraint violation in the DBMS.
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.	Results were pending on the *StatementHandle* from a previous query or a cursor associated with the *hsmt* had not been closed.

Table 58. SQLExecDirect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
34000	Invalid cursor name.	*StatementText* contained a Positioned DELETE or a Positioned UPDATE and the cursor referenced by the statement being executed was not open.
37xxx [a]	Invalid SQL syntax.	*StatementText* contained one or more of the following: • a COMMIT • a ROLLBACK • an SQL statement that the connected database server could not prepare • a statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belonged was rolled back due to a deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
42xxx	Syntax Error or Access Rule Violation.	**42**5xx indicates the authorization ID does not have permission to execute the SQL statement contained in *StatementText*. Other **42**xxx SQLSTATES indicate a variety of syntax or access problems with the statement.

Table 58. SQLExecDirect SQLSTATEs (continued)

SQLSTATE	Description	Explanation

428A1	Unable to access a file referenced by a host file variable.	This can be raised for any of the following scenarios. The associated reason code in the text identifies the particular error:
		• 01 - The file name length is invalid or the file name and/or the path has an invalid format.
		• 02 - The file option is invalid. It must have one of the following values:
		`SQL_FILE_READ -read from an existing file`
		`SQL_FILE_CREATE -create a new file for write`
		`SQL_FILE_OVERWRITE -overwrite an existing file. If the file does not exist, create the file.`
		`SQL_FILE_APPEND -append to an existing file. If the file does not exist, create the file.`
		• 03 - The file cannot be found.
		• 04 - The SQL_FILE_CREATE option was specified for a file with the same name as an existing file.
		• 05 - Access to the file was denied. The user does not have permission to open the file.
		• 06 - Access to the file was denied. The file is in use with incompatible modes. Files to be written to are opened in exclusive mode.
		• 07 - Disk full was encountered while writing to the file.
		• 08 - Unexpected end of file encountered while reading from the file.
		• 09 - A media error was encountered while accessing the file.

42895	The value of a host variable in the EXECUTE or OPEN statement cannot be used because of its data type.	The LOB locator type specified on the bind parameter function call does not match the LOB data type of the parameter marker. The argument *fSQLType* used on the bind parameter function specified a LOB locator type but the corresponding parameter marker is not a LOB.
44000	Integrity constraint violation.	*StatementText* contained an SQL statement which contained a parameter or literal. This parameter value was NULL for a column defined as NOT NULL in the associated table column, or a duplicate value was supplied for a column constrained to contain only unique values, or some other integrity constraint was violated.

Table 58. SQLExecDirect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
56084	LOB data is not supported in DRDA.	LOB columns cannot either be selected or updated when connecting to DRDA servers (using DB2 Connect).
58004	Unexpected system failure.	Unrecoverable system error.
S0001	Database object already exists.	*StatementText* contained a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already existed.
S0002	Database object does not exist.	*StatementText* contained an SQL statement that references a table name or view name which does not exist.
S0011	Index already exists.	*StatementText* contained a CREATE INDEX statement and the specified index name already existed.
S0012	Index not found.	*StatementText* contained a DROP INDEX statement and the specified index name did not exist.
S0021	Column already exists.	*StatementText* contained an ALTER TABLE statement and the column specified in the ADD clause was not unique or identified an existing column in the base table.
S0022	Column not found.	*StatementText* contained an SQL statement that references a column name which does not exist.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY009	Invalid argument value.	*StatementText* was a null pointer.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The argument *TextLength* was less than 1 but not equal to SQL_NTS.
HY092	Option type out of range.	The *FileOptions* argument of a previous `SQLBindFileToParam()` operation was not valid.
HY503	Invalid file name length.	The *fileNameLength* argument value from `SQLBindFileToParam()` was less than 0, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Table 58. SQLExecDirect SQLSTATEs (continued)

SQLSTATE	Description	Explanation

Note:

a xxx refers to any SQLSTATE with that class code. Example, **37**xxx refers to any SQLSTATE in the **37** class.

Restrictions

None.

Example

Refer to “Example” on page 406.

References

- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLBindFileToCol - Bind LOB File Reference to LOB Column” on page 237
- “SQLBindFileToParam - Bind LOB File Reference to LOB Parameter” on page 242
- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247
- “SQLExecute - Execute a Statement” on page 373
- “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408
- “SQLFetch - Fetch Next Row” on page 396
- “SQLParamData - Get Next Parameter For Which A Data Value Is Needed” on page 575
- “SQLPutData - Passing Data Value for A Parameter” on page 609
- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247

SQLExecute - Execute a Statement

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLExecute() executes a statement, that was successfully prepared using SQLPrepare(), once or multiple times. The statement is executed using the current value of any application variables that were bound to parameter markers by SQLBindParameter(), SQLSetParam() or SQLBindFileToParam()

Syntax

```
SQLRETURN   SQLExecute      (SQLHSTMT          StatementHandle);  /* hstmt */
```

Function Arguments

Table 59. SQLExecute Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle. There must not be an open cursor associated with StatementHandle, see “SQLFreeStmt - Free (or Reset) a Statement Handle” on page 435 for more information.

Usage

The SQL statement string may contain parameter markers. A parameter marker is represented by a ″?″ character, and is used to indicate a position in the statement where an application supplied value is to be substituted when SQLExecute() is called. This value can be obtained from:

- An application variable.

  SQLSetParam() or SQLBindParameter() is used to bind the application storage area to the parameter marker.
- A LOB value residing at the server referenced by a LOB locator. SQLBindParameter() or SQLSetParam() is used to bind a LOB locator to a parameter marker. The actual value of the LOB is kept at the server and does not need to be first transferred to the application before being used as input parameter value for another SQL statement.
- A file (within the applications environment) containing a LOB value.

  SQLBindFileToParam() is used to bind a file to a LOB parameter marker. When SQLExecDirect() is executed, DB2 CLI will transfer the contents of the file directly to the database server.

All parameters must be bound before calling SQLExecute().

SQLExecute

Once the application has processed the results from the SQLExecute() call, it can execute the statement again with new (or the same) parameter values.

A statement executed by SQLExecDirect() cannot be re-executed by calling SQLExecute(); SQLPrepare() must be called first.

If the prepared SQL statement is a query, SQLExecute() will generate a cursor name, and open the cursor. If the application has used SQLSetCursorName() to associate a cursor name with the statement handle, DB2 CLI associates the application generated cursor name with the internally generated one.

To execute a query more than once, the application must close the cursor by calling SQLFreeStmt() with the SQL_CLOSE option. There must not be an open cursor on the statement handle when calling SQLExecute().

If a result set is generated, SQLFetch() or SQLFetchScroll() will retrieve the next row (or rows) of data into bound variables, LOB locators or LOB file references (using SQLBindCol() or SQLBindFileToCol). Data can also be retrieved by calling SQLGetData() for any column that was not bound.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement must be positioned on a row at the time SQLExecute() is called, and must be defined on a separate statement handle under the same connection handle.

If SQLParamOptions() has been called to specify that an array of input parameter values has been bound to each parameter marker, then the application needs to call SQLExecDirect() only once to process the entire array of input parameter values. If the executed statement returns multiple result sets (one for each set of input parameters), then SQLMoreResults() should be used to advance to the next result set once processing on the current result set is complete. Refer to “SQLMoreResults - Determine If There Are More Result Sets” on page 562 for more information.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA
- SQL_NO_DATA_FOUND

SQL_NEED_DATA is returned when the application has requested to input data-at-execution parameter values by calling SQLParamData() and SQLPutData().

SQL_NO_DATA_FOUND is returned if the SQL statement is a Searched UPDATE or Searched DELETE and no rows satisfy the search condition.

Diagnostics

The SQLSTATEs for SQLExecute() include all those for SQLExecDirect() (refer to Table 58 on page 367.) except for **HY**009, **HY**090 and with the addition of the SQLSTATE in the table below.

Table 60. SQLExecute SQLSTATEs

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The specified *StatementHandle* was not in prepared state. SQLExecute() was called without first calling SQLPrepare().

Authorization

None.

Example

Refer to “Example” on page 406.

References

- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLExecute - Execute a Statement” on page 373
- “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408
- “SQLPrepare - Prepare a Statement” on page 583
- “SQLFetch - Fetch Next Row” on page 396
- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247
- “SQLParamOptions - Specify an Input Array for a Parameter” on page 579
- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247
- “SQLBindFileToParam - Bind LOB File Reference to LOB Parameter” on page 242
- “SQLBindFileToCol - Bind LOB File Reference to LOB Column” on page 237

- "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227

SQLExtendedBind - Bind an Array of Columns

Purpose

Specification:	DB2 CLI 6		

SQLExtendedBind() is used to bind an array of columns instead of using repeated calls to SQLBindCol() or SQLBindParameter().

Syntax

```
SQLRETURN    SQLExtendedBind (
                              SQLHSTMT          StatementHandle,
                              SQLSMALLINT       fBindCol,
                              SQLSMALLINT       cRecords,
                              SQLSMALLINT *     pfCType,
                              SQLPOINTER  *     rgbValue,
                              SQLINTEGER  *     cbValueMax,
                              SQLUINTEGER *     puiPrecisionCType,
                              SQLSMALLINT *     psScaleCType,
                              SQLINTEGER  **    pcbValue,
                              SQLINTEGER  **    piIndicator,
                              SQLSMALLINT *     pfParamType,
                              SQLSMALLINT *     pfSQLType,
                              SQLUINTEGER *     pcbColDef,
                              SQLSMALLINT *     pibScale ) ;
```

Function Arguments

Table 61. SQLExtendedBind() Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLSMALLINT	fBindCol	input	If SQL_TRUE then this output information similar to SQLBindCol(), otherwise, it is similar to SQLBindParameter().
SQLSMALLINT	cRecords	input	Number of columns to bind.
SQLSMALLINT	pfCType	input	Array of values for the application data type.
SQLPOINTER	rgbValue	input	Array of pointers to application data area.
SQLINTEGER	cbValueMax	input	Array of maximum sizes for *rgbValue*.
SQLUINTEGER	puiPrecisionCType	input	Decimal precision of record. Only used if the application data type is SQL_C_DECIMAL_IBM.
SQLSMALLINT	psScaleCType	input	Decimal scale of record. Only used if the application data type is SQL_C_DECIMAL_IBM.
SQLINTEGER	pcbValue	input	Array of pointers to length values.
SQLINTEGER	piIndicator	input	Array of pointers to indicator values.

Table 61. SQLExtendedBind() Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	pfParamType	input	Array of parameter types. Only used if fBindCol is FALSE. Each row in this array serves the same purpose as the SQLBindParameter() argument *InputOutputType*. It can be set to: • SQL_PARAM_INPUT • SQL_PARAM_INPUT_OUTPUT • SQL_PARAM_OUTPUT
SQLSMALLINT	pfSQLType	input	Array of SQL data types. Only used if fBindCol is FALSE. Each row in this array serves the same purpose as the SQLBindParameter() argument *ParameterType*.
SQLUINTEGER	pcbColDef	input	Array of SQL precision values. Only used if *fBindCol* is FALSE. Each row in this array serves the same purpose as the SQLBindParameter() argument *ColumnSize*.
SQLSMALLINT	pibScale	input	Array of SQL scale values. Only used if *fBindCol* is FALSE. Each row in this array serves the same purpose as the SQLBindParameter() argument *DecimalDigits*.

Usage

This function is used to replace multiple calls to SQLBindCol() or SQLBindParameter().

The argument *fBindCol* determines whether this function call is used to associate (bind):

- parameter markers in an SQL statement (as with SQLBindParameter()) - *fBindCol* = SQL_FALSE
- columns in a result set (as with SQLBindCol()) - *fBindCol* = SQL_TRUE

For more information (including allowed argument values), see “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227 and “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 62. SQLExtendedBind() SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The conversion from the data value identified by a row in the *pfCType* argument to the data type identified by the *pfParamType* argument is not a meaningful conversion. (For example, conversion from SQL_C_DATE to SQL_DOUBLE.)
07009	Invalid descriptor index	The value specified for the argument *cRecords* exceeded the maximum number of columns in the result set.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	A row in *pfParamType* or *pfSQLType* was not a valid data type or SQL_C_DEFAULT.
HY004	SQL data type out of range.	The value specified for the argument *pfParamType* is not a valid SQL data type.
HY009	Invalid argument value.	The argument *rgbValue* was a null pointer and the argument *cbValueMax* was a null pointer, and *pfParamType* is not SQL_PARAM_OUTPUT.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY021	Inconsistent descriptor information	The descriptor information checked during a consistency check was not consistent.
HY090	Invalid string or buffer length.	The value specified for the argument *cbValueMax* is less than 1 and the argument the corresponding row in *pfParamType* or *pfSQLType* is either SQL_C_CHAR, SQL_C_BINARY or SQL_C_DEFAULT.

Table 62. SQLExtendedBind() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY093	Invalid parameter number.	The value specified for a row in the argument *pfCType* was less than 1 or greater than the maximum number of parameters supported by the server.
HY094	Invalid scale value.	The value specified for *pfParamType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified for *DecimalDigits* was less than 0 or greater than the value for the argument *pcbColDef* (precision). The value specified for *pfParamType* was SQL_C_TIMESTAMP and the value for *pfParamType* was either SQL_CHAR or SQL_VARCHAR and the value for *DecimalDigits* was less than 0 or greater than 6.
HY104	Invalid precision value.	The value specified for *pfParamType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified by *pcbColDef* was less than 1.
HY105	Invalid parameter type.	*pfParamType* is not one of SQL_PARAM_INPUT, SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT.
HYC00	Driver not capable.	DB2 CLI recognizes, but does not support the data type specified in the row in *pfParamType* or *pfSQLType*. A LOB locator C data type was specified, but the connected server does not support LOB data types.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227
- "SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator" on page 247

SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)

Deprecated

Note:

In ODBC version 3, SQLExtendedFetch() has been deprecated and replaced with SQLFetchScroll(); see "SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns" on page 408 for more information.

Although this version of DB2 CLI continues to support SQLExtendedFetch(), we recommend that you begin using SQLFetchScroll() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Purpose

Specification:	**DB2 CLI** 2.1	**ODBC** 1.0	

SQLExtendedFetch() extends the function of SQLFetch() by returning a block of data containing multiple rows (called a *rowset*), in the form of a array, for each bound column. The size of the rowset is determined by the SQL_ROWSET_SIZE attribute on an SQLSetStmtAttr() call.

To fetch one row of data at a time, an application should call SQLFetch().

For more description on block or array retrieval, refer to "Retrieving a Result Set into an Array" on page 90.

Syntax

```
SQLRETURN   SQLExtendedFetch (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLUSMALLINT      FetchOrientation,  /* fFetchType */
                SQLINTEGER        FetchOffset,       /* irow */
                SQLUINTEGER  *FAR RowCountPtr,       /* pcrow */
                SQLUSMALLINT *FAR RowStatusArray);   /* rgfRowStatus */
```

Function Arguments

Table 63. SQLExtendedFetch Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.

Table 63. SQLExtendedFetch Arguments (continued)

Data Type	Argument	Use	Description
SQLUSMALLINT	FetchOrientation	Input	Direction and type of fetch. DB2 CLI only supports the fetch direction SQL_FETCH_NEXT; that is, forward only cursor direction. The next array (rowset) of data is retrieved.
SQLINTEGER	FetchOffset	Input	Reserved for future use.
SQLUINTEGER *	RowCountPtr	Output	Number of the rows actually fetched. If an error occurs during processing, *RowCountPtr* points to the ordinal position of the row (in the rowset) that precedes the row where the error occurred. If an error occurs retrieving the first row *RowCountPtr* points to the value 0.
SQLUSMALLINT *	RowStatusArray	Output	An array of status values. The number of elements must equal the number of rows in the rowset (as defined by the SQL_ROWSET_SIZE attribute). A status value for each row fetched is returned: • SQL_ROW_SUCCESS If the number of rows fetched is less than the number of elements in the status array (i.e. less than the rowset size), the remaining status elements are set to SQL_ROW_NOROW. DB2 CLI cannot detect whether a row has been updated or deleted since the start of the fetch. Therefore, the following ODBC defined status values will not be reported: • SQL_ROW_DELETED • SQL_ROW_UPDATED

Usage

SQLExtendedFetch() is used to perform an array fetch of a set of rows. An application specifies the size of the array by calling SQLSetStmtAttr() with the SQL_ROWSET_SIZE attribute.

Before SQLExtendedFetch() is called the first time, the cursor is positioned before the first row. After SQLExtendedFetch() is called, the cursor is positioned on the row in the result set corresponding to the last row element in the rowset just retrieved.

For any columns in the result set that have been bound via the SQLBindCol() or SQLBindFileToCol() function, DB2 CLI converts the data for the bound

columns as necessary and stores it in the locations bound to these columns. As mentioned in section “Retrieving a Result Set into an Array” on page 90, the result set can be bound in a column-wise or row-wise fashion.

- For column-wise binding of application variables:

  To bind a result set in column-wise fashion, an application specifies SQL_BIND_BY_COLUMN for the SQL_ATTR_BIND_TYPE statement attribute. (This is the default value.) Then the application calls the SQLBindCol() function.

  When the application calls SQLExtendedFetch(), data for the first row is stored at the start of the buffer. Each subsequent row of data is stored at an offset of *cbValueMax* bytes (argument on SQLBindCol() call) or, if the associated C buffer type is fixed width (such as SQL_C_LONG), at an offset corresponding to that fixed length from the data for the previous row.

  For each bound column, the number of bytes available to return for each element is stored in the *pcbValue* array buffer (deferred output argument on SQLBindCol()) buffer bound to the column. The number of bytes available to return for the first row of that column is stored at the start of the buffer, and the number of bytes available to return for each subsequent row is stored at an offset of *sizeof(SQLINTEGER)* bytes from the value for the previous row. If the data in the column is NULL for a particular row, the associated element in the pcbValue array is set to SQL_NULL_DATA.

- For column-wise binding of file references:

  The *StringLength* and *IndicatorValue* pointers on SQLBindFileToCol() are pointers to output arrays. The actual length of the file and the associated indicator value for the first row is stored at the start of the StringLength and IndicatorValue arrays respectively. File lengths and indicator values for subsequent rows are written to these arrays at an offset of *sizeof(SQLINTEGER)* bytes from the previous row.

- For row-wise binding of application variables:

  The application needs to first call SQLSetStmtAttr() with the SQL_ATTR_BIND_TYPE attribute, with the *vParam* argument set to the size of the structure capable of holding a single row of retrieved data and the associated data lengths for each column data value.

  For each bound column, the first row of data is stored at the address given by the *rgbValue* supplied on the SQLBindCol() call for the column and each subsequent row of data at an offset of *vParam* bytes (used on the SQLSetStmtAttr() call) from the data for the previous row.

  For each bound column, the number of bytes available to return for the first row is stored at the address given by the *pcbValue* argument supplied on the SQLBindCol() call, and the number of bytes available to return for each subsequent row at an offset of *vParam* bytes from address containing the value for the previous row.

Row-wise binding of file references is not supported.

If SQLExtendedFetch() returns an error that applies to the entire rowset, the SQL_ERROR function return code is reported with the appropriate SQLSTATE. The contents of the rowset buffer are undefined and the cursor position is unchanged.

If an error occurs that applies to a single row:

- the corresponding element in the *RowStatusArray* array for the row is set to SQL_ROW_ERROR
- an SQLSTATE of **01**S01 is added to the list of errors that can be obtained using *SQLError()*
- zero or more additional SQLSTATEs, describing the error for the current row, are added to the list of errors that can be obtained using *SQLError()*

An SQL_ROW_ERROR in the *RowStatusArray* array only indicates that there was an error with the corresponding element; it does not indicate how many SQLSTATEs were generated. Therefore, SQLSTATE **01**S01 is used as a separator between the resulting SQLSTATEs for each row. DB2 CLI continues to fetch the remaining rows in the rowset and returns SQL_SUCCESS_WITH_INFO as the function return code. After SQLExtendedFetch() returns, for each row encountering an error there is an SQLSTATE of **01**S01 and zero or more additional SQLSTATEs indicating the error(s) for the current row, retrievable via SQLError(). Individual errors that apply to specific rows do not affect the cursor which continues to advance.

The number of elements in the *RowStatusArray* array output buffer must equal the number of rows in the rowset (as defined by the SQL_ROWSET_SIZE statement attribute). If the number of rows fetched is less than the number of elements in the status array, the remaining status elements are set to SQL_ROW_NOROW.

An application cannot mix SQLExtendedFetch() with SQLFetch() calls.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

Diagnostics

Table 64. SQLExtendedFetch SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data returned for one *or more* columns was truncated. (Function returns SQL_SUCCESS_WITH_INFO.)
01S01	Error in row.	An error occurred while fetching one or more rows. (Function returns SQL_SUCCESS_WITH_INFO.)
07002	Too many columns.	A column number specified in the binding for one or more columns was greater than the number of columns in the result set.
07006	Invalid conversion.	The data value could not be converted in a meaningful manner to the data type specified by *fCType* in `SQLBindCol()`.
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument *pcbValue* in `SQLBindCol()` was a null pointer and the value of the corresponding column is null. There is no means to report SQL_NULL_DATA. The pointer specified for the argument *IndicatorValue* in `SQLBindFileToCol()` was a null pointer and the value of the corresponding LOB column is NULL. There is no means to report SQL_NULL_DATA.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for one or more columns would have caused the whole part of the number to be truncated either at the time of assignment or in computing an intermediate result. A value from an arithmetic expression was returned which resulted in division by zero. **Note:** The associated cursor is undefined if this error is detected by DB2 Universal Database. If the error was detected by DB2 CLI or by other IBM RDBMSs, the cursor will remain open and continue to advance on subsequent fetch calls.
22005	Error in assignment.	A returned value was incompatible with the data type of the bound column. A returned LOB locator was incompatible with the data type of the bound column.
22007	Invalid datetime format.	Conversion from character a string to a datetime format was indicated, but an invalid string representation or value was specified, or the value was an invalid date. The value of a date, time, or timestamp does not conform to the syntax for the specified data type.

Table 64. SQLExtendedFetch SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	A value from an arithmetic expression was returned which resulted in division by zero.
24000	Invalid cursor state.	The previous SQL statement executed on the statement handle was not a query.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.

428A1	Unable to access a file referenced by a host file variable.	This can be raised for any of the following scenarios. The associated reason code in the text identifies the particular error:
		• 01 - The file name length is invalid or the file name and/or the path has an invalid format.
		• 02 - The file option is invalid. It must have one of the following values:
		`SQL_FILE_READ -read from an existing file`
		`SQL_FILE_CREATE -create a new file for write`
		`SQL_FILE_OVERWRITE -overwrite an existing file. If the file does not exist, create the file.`
		`SQL_FILE_APPEND -append to an existing file. If the file does not exist, create the file.`
		• 03 - The file cannot be found.
		• 04 - The SQL_FILE_CREATE option was specified for a file with the same name as an existing file.
		• 05 - Access to the file was denied. The user does not have permission to open the file.
		• 06 - Access to the file was denied. The file is in use with incompatible modes. Files to be written to are opened in exclusive mode.
		• 07 - Disk full was encountered while writing to the file.
		• 08 - Unexpected end of file encountered while reading from the file.
		• 09 - A media error was encountered while accessing the file.

Table 64. SQLExtendedFetch SQLSTATEs (continued)

SQLSTATE	Description	Explanation
54028	The maximum number of concurrent LOB handles has been reached.	Maximum LOB locator assigned. The maximum number of concurrent LOB locators has been reached. A new locator can not be assigned.
56084	LOB data is not supported in DRDA.	LOBs not supported on DRDA. LOB columns cannot either be selected or updated when connecting to DRDA servers (using DB2 Connect).
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	`SQLExtendedFetch()` was called for an StatementHandle after `SQLFetch()` was called and before `SQLFreeStmt()` had been called with the SQL_CLOSE option. The function was called prior to calling `SQLPrepare()` or `SQLExecDirect()` for the *StatementHandle*. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY092	Option type out of range.	The *FileOptions* argument of a previous `SQLBindFileToCol()` operation was not valid.
HY106	Fetch type out of range.	The value specified for the argument *FetchOrientation* was not recognized.
HYC00	Driver not capable.	DB2 CLI or the data source does not support the conversion specified by the combination of the *fCType* in `SQLBindCol()` or `SQLBindFileToCol()` and the SQL data type of the corresponding column. A call to `SQLBindCol()` was made for a column data type which is not supported by DB2 CLI. The specified fetch type is recognized, but not supported.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Restrictions

None.

Example

Refer to “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408.

References

- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLBindFileToCol - Bind LOB File Reference to LOB Column” on page 237
- “SQLExecute - Execute a Statement” on page 373
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLFetch - Fetch Next Row” on page 396

SQLExtendedPrepare - Prepare a Statement and Set Statement Attributes

Purpose

Specification:	DB2 CLI 6.0		

SQLExtendedPrepare() is used to prepare a statement and set a group of statement attributes, all in one call.

This function can be used in place of a call to SQLPrepare() followed by a number of calls to SQLSetStmtAttr().

Syntax

```
SQLRETURN SQLExtendedPrepare( SQLHSTMT      StatementHandle,
                      SQLCHAR *     StatementText,
                      SQLINTEGER    TextLength,
                      SQLINTEGER    cPars,
                      SQLSMALLINT   sStmtType,
                      SQLINTEGER    cStmtAttrs,
                      SQLINTEGER *  piStmtAttr,
                      SQLINTEGER *  pvParams );
```

Function Arguments

Table 65. SQLExtendedPrepare() Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.
SQLCHAR	StatementText	Input	SQL statement string.
SQLINTEGER	TextLength	Input	Length of the contents of the *StatementText* argument. This must be set to either the exact length of the SQL statement in *StatementText*, or to SQL_NTS if the statement text is null-terminated.
SQLINTEGER	cPars	Input	Number of parameter markers in statement.
SQLSMALLINT	sStmtType	Input	Statement type. For possible values see “List of cStmtType Values” on page 390.
SQLINTEGER	cStmtAttrs	Input	Number of statement attributes specified on this call.
SQLINTEGER	piStmtAttr	Input	Array of statement attributes to set.
SQLINTEGER	pvParams	Input	Array of corresponding statement attributes values to set.

SQLExtendedPrepare

Usage

The first three arguments of this function are exactly the same as the arguments in `SQLPrepare()`.

There are two requirements when using `SQLExtendedBind()`:

1. The SQL statements will not be scanned for ODBC/vendor escape clauses. It behaves as if the SQL_ATTR_NOSCAN statment attribute is set to SQL_NOSCAN. If the SQL statement contains ODBC/vendor escape clauses then `SQLExtendedBind()` cannot be used.
2. You must indicate in advance (through *cRecords*) the number of parameter markers that are included in the SQL statement.

For more information on *StatementHandle*, *StatementText*, and *TextLength* see "SQLPrepare - Prepare a Statement" on page 583.

The *cPars* argument indicates the number of parameter markers in *StatementText*.

The argument *cStmtType* is used to indicate the type of statement that is being prepare. See "List of cStmtType Values" for the list of possible values.

The final three arguments are used to indicate the set of statement attributes to use. Set *cStmtAttrs* to the number of statement attributes specified on this call. Create two arrays, one to hold the list of statement attributes, one to hold the value for each. Use these arrays for *piStmtAttr* and *pvParams*. For details on the possible statement attributes, see "SQLSetStmtAttr - Set Options Related to a Statement" on page 702.

List of cStmtType Values

The argument *cStmtType* can be set to one of the following values:

- SQL_CLI_STMT_UNDEFINED
- SQL_CLI_STMT_ALTER_TABLE
- SQL_CLI_STMT_CREATE_INDEX
- SQL_CLI_STMT_CREATE_TABLE
- SQL_CLI_STMT_CREATE_VIEW
- SQL_CLI_STMT_DELETE_SEARCHED
- SQL_CLI_STMT_DELETE_POSITIONED
- SQL_CLI_STMT_GRANT
- SQL_CLI_STMT_INSERT
- SQL_CLI_STMT_REVOKE

- SQL_CLI_STMT_SELECT
- SQL_CLI_STMT_UPDATE_SEARCHED
- SQL_CLI_STMT_UPDATE_POSITIONED
- SQL_CLI_STMT_CALL
- SQL_CLI_STMT_SELECT_FOR_UPDATE
- SQL_CLI_STMT_WITH
- SQL_CLI_STMT_SELECT_FOR_FETCH
- SQL_CLI_STMT_VALUES
- SQL_CLI_STMT_CREATE_TRIGGER
- SQL_CLI_STMT_SELECT_INTO
- SQL_CLI_STMT_CREATE_PROCEDURE
- SQL_CLI_STMT_CREATE_FUNCTION
- SQL_CLI_STMT_SET_CURRENT_QUERY_OPT

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 66. SQLExtendedPrepare SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01504	The UPDATE or DELETE statement does not include a WHERE clause.	*StatementText* contained an UPDATE or DELETE statement which did not contain a WHERE clause.
01508	Statement disqualified for blocking.	The statement was disqualified for blocking for reasons other than storage.
01S02	Option value changed.	DB2 CLI did not support a value specified in **pvParams*, or a value specified in **pvParams* was invalid because of SQL constraints or requirements, so DB2 CLI substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was connected failed before the function completed processing.

Table 66. SQLExtendedPrepare SQLSTATEs (continued)

SQLSTATE	Description	Explanation
21S01	Insert value list does not match column list.	*StatementText* contained an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*StatementText* contained a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
22018	Invalid character value for cast specification.	**StatementText* contained an SQL statement that contained a literal or parameter and the value was incompatible with the data type of the associated table column.
22019	Invalid escape character	The argument *StatementText* contained a LIKE predicate with an ESCAPE in the WHERE clause, and the length of the escape character following ESCAPE was not equal to 1.
22025	Invalid escape sequence	The argument *StatementText* contained "LIKE *pattern value* ESCAPE *escape character*" in the WHERE clause, and the character following the escape character in the pattern value was not one of "%" or "_".
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
34000	Invalid cursor name.	*StatementText* contained a Positioned DELETE or a Positioned UPDATE and the cursor referenced by the statement being executed was not open.
37xxx [a]	Invalid SQL syntax.	*StatementText* contained one or more of the following: • a COMMIT • a ROLLBACK • an SQL statement that the connected database server could not prepare • a statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belonged was rolled back due to deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
42xxx [a]	Syntax Error or Access Rule Violation.	**42**5xx indicates the authorization ID does not have permission to execute the SQL statement contained in *StatementText*. Other **42**xxx SQLSTATES indicate a variety of syntax or access problems with the statement.
58004	Unexpected system failure.	Unrecoverable system error.
S0001	Database object already exists.	*StatementText* contained a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already existed.

Table 66. SQLExtendedPrepare SQLSTATEs (continued)

SQLSTATE	Description	Explanation
S0002	Database object does not exist.	*StatementText* contained an SQL statement that references a table name or a view name which did not exist.
S0011	Index already exists.	*StatementText* contained a CREATE INDEX statement and the specified index name already existed.
S0012	Index not found.	*StatementText* contained a DROP INDEX statement and the specified index name did not exist.
S0021	Column already exists.	*StatementText* contained an ALTER TABLE statement and the column specified in the ADD clause was not unique or identified an existing column in the base table.
S0022	Column not found.	*StatementText* contained an SQL statement that references a column name which did not exist.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY009	Invalid argument value.	*StatementText* was a null pointer.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY011	Operation invalid at this time.	The *Attribute* was SQL_ATTR_CONCURRENCY, SQL_ ATTR_CURSOR_TYPE, SQL_ ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS and the statement was prepared.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.

Table 66. SQLExtendedPrepare SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY017	Invalid use of an automatically allocated descriptor handle.	The *Attribute* argument was SQL_ATTR_IMP_ROW_DESC or SQL_ATTR_IMP_PARAM_DESC. The *Attribute* argument was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and the value in **ValuePtr* was an implicitly allocated descriptor handle.
HY024	Invalid attribute value.	Given the specified *Attribute* value, an invalid value was specified in **ValuePtr*. (DB2 CLI returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE or SQL_ATTR_ASYNC_ENABLE. For all other connection and statement attributes, the driver must verify the value specified in **ValuePtr*.)
HY090	Invalid string or buffer length.	The argument *TextLength* was less than 1, but not equal to SQL_NTS.
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid for this version of DB2 CLI.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid connection or statement attribute for the version of the DB2 CLI driver, but was not supported by the data source.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Note:

a xxx refers to any SQLSTATE with that class code. Example, **37**xxx refers to any SQLSTATE in the **37** class.

Note: Not all DBMSs report all of the above diagnostic messages at prepare time. Therefore an application must also be able to handle these conditions when calling `SQLExecute()`.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- “SQLPrepare - Prepare a Statement” on page 583
- “SQLSetStmtAttr - Set Options Related to a Statement” on page 702

SQLFetch - Fetch Next Row

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLFetch() advances the cursor to the next row of the result set, and retrieves any bound columns.

Columns may be bound to:

- Application storage
- LOB locators
- Lob file references

When SQLFetch() is called, the appropriate data transfer is performed, along with any data conversion if conversion was indicated when the column was bound. The columns can also be received individually after the fetch, by calling SQLGetData().

SQLFetch() can only be called after a result set has been generated (using the same statement handle) by either executing a query, calling SQLGetTypeInfo() or calling a catalog function.

To retrieve multiple rows at a time, use SQLFetchScroll().

Syntax

```
SQLRETURN   SQLFetch        (SQLHSTMT         StatementHandle);  /* hstmt */
```

Function Arguments

Table 67. SQLFetch Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle

Usage

SQLFetch() can only be called after a result set has been generated on the same statement handle. Before SQLFetch() is called the first time, the cursor is positioned before the start of the result set.

The number of application variables bound with SQLBindCol() must not exceed the number of columns in the result set or SQLFetch() will fail.

If SQLBindCol() has not been called to bind any columns, then SQLFetch() does not return data to the application, but just advances the cursor. In this

case SQLGetData() could be called to obtain all of the columns individually. If the cursor is a multirow cursor (that is, the SQL_ATTR_ROW_ARRAY_SIZE is greater than 1), SQLGetData() can be called only if SQL_GD_BLOCK is returned when SQLGetInfo() is called with an *InfoType* of SQL_GETDATA_EXTENSIONS. Data in unbound columns is discarded when SQLFetch() advances the cursor to the next row. For fixed length data types, or small variable length data types, binding columns provides better performance than using SQLGetData().

Columns may be bound to:

- Application storage

  SQLBindCol() is used to bind application storage to the column. Data will be transferred from the server to the application at fetch time. Length of the available data to return is also set.

- LOB locators

  SQLBindCol() is used to bind LOB locators to the column. Only the LOB locator (4 bytes) will be transferred from the server to the application at fetch time.

  Once an application receives a locator it can be used in SQLGetSubString(), SQLGetPosition(), SQLGetLength() or as the value of a parameter marker in another SQL statement. SQLGetSubString() can either return another locator, or the data itself. All locators remain valid until the end of the transaction in which they were created (even when the cursor moves to another row), or until it is freed using the FREE LOCATOR statement.

- Lob file references

  SQLBindFileToCol() is used to bind a file to a LOB column. DB2 CLI will write the data directly to a file, and update the StringLength and IndicatorValue buffers specified on SQLBindFileToCol().

  If the data value for the column is NULL and SQLBindCol() was used, SQL_NULL_DATA is stored in the *pcbValue* buffer specified on SQLBindCol().

  If the data value for the column is NULL and SQLBindFileToCol() was used, then *IndicatorValue* will be set to SQL_NULL_DATA and *StringLength* to 0.

If LOB values are too large to be retrieved in one fetch, they can be retrieved in pieces by either using SQLGetData() (which can be used for any column type), or by binding a LOB locator, and using SQLGetSubString().

If any bound storage buffer are not large enough to hold the data returned by SQLFetch(), the data will be truncated. If character data is truncated, SQL_SUCCESS_WITH_INFO is returned, and an SQLSTATE is generated indicating truncation. The SQLBindCol() deferred output argument *pcbValue* will contain the actual length of the column data retrieved from the server.

The application should compare the actual output length to the input buffer length (*pcbValue* and *cbValueMax* arguments from SQLBindCol()) to determine which character columns have been truncated.

Truncation of numeric data types is reported as a warning if the truncation involves digits to the right of the decimal point. If truncation occurs to the left of the decimal point, an error is returned (refer to the diagnostics section).

Truncation of graphic data types is treated the same as character data types, except that the *rgbValue* buffer is filled to the nearest multiple of two bytes that is still less than or equal to the *cbValueMax* specified in SQLBindCol(). Graphic (DBCS) data transferred between DB2 CLI and the application is not null-terminated if the C buffer type is SQL_C_CHAR (unless indicated by the PATCH1 initialization keyword, refer to "Configuration Keywords" on page 164 for more information). If the buffer type is SQL_C_DBCHAR, then null-termination of graphic data does occur.

Truncation is also affected by the SQL_ATTR_MAX_LENGTH statement attribute. The application can specify that DB2 CLI should not report truncation by calling SQLSetStmtAttr() with SQL_ATTR_MAX_LENGTH and a value for the maximum length to return for any one column, and by allocating a *rgbValue* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS will be returned and the maximum length, not the actual length will be returned in *pcbValue*.

When all the rows have been retrieved from the result set, or the remaining rows are not needed, SQLFreeStmt() should be called to close the cursor and discard the remaining data and associated resources.

To retrieve multiple rows at a time, use SQLFetchScroll(). An application cannot mix SQLFetch() with SQLExtendedFetch() calls on the same statement handle. It can, however, mix SQLFetch() with SQLFetchScroll() calls on the same statement handle.

Positioning the Cursor

When the result set is created, the cursor is positioned before the start of the result set. SQLFetch() fetches the next rowset. It is equivalent to calling SQLFetchScroll() with *FetchOrientation* set to SQL_FETCH_NEXT. For more information see "Scrollable Cursors" on page 68.

The SQL_ATTR_ROW_ARRAY_SIZE statement attribute specifies the number of rows in the rowset. If the rowset being fetched by SQLFetch() overlaps the end of the result set, SQLFetch() returns a partial rowset. That is, if S + R-1 is greater than L, where S is the starting row of the rowset being fetched, R is

the rowset size, and L is the last row in the result set, then only the first L-S+1 rows of the rowset are valid. The remaining rows are empty and have a status of SQL_ROW_NOROW.

See SQLFetchScroll(), "Cursor Positioning Rules" on page 409 under SQL_FETCH_NEXT for more information.

After SQLFetch() returns, the current row is the first row of the rowset.

Row Status Array

SQLFetch() sets values in the row status array in the same manner as SQLFetchScroll(). For more information see SQLFetchScroll(), "Row Status" on page 413.

Rows Fetched Buffer

SQLFetch() returns the number of rows fetched in the rows fetched buffer in the same manner as SQLFetchScroll(). For more information see SQLFetchScroll(), "Rows Fetched Buffer" on page 414.

Error Handling

Errors and warnings can apply to individual rows or to the entire function. For more information about diagnostic records see "SQLGetDiagField - Get a Field of Diagnostic Data" on page 468.

Errors and Warnings on the Entire Function

If an error applies to the entire function, such as SQLSTATE HYT00 (Timeout expired) or SQLSTATE 24000 (Invalid cursor state), SQLFetch() returns SQL_ERROR and the applicable SQLSTATE. The contents of the rowset buffers are undefined and the cursor position is unchanged.

If a warning applies to the entire function, SQLFetch() returns SQL_SUCCESS_WITH_INFO and the applicable SQLSTATE. The status records for warnings that apply to the entire function are returned before the status records that apply to individual rows.

Errors and Warnings in Individual Rows

If an error (such as SQLSTATE 22012 (Division by zero)) or a warning (such as SQLSTATE 01004 (Data truncated)) applies to a single row, SQLFetch():

- Sets the corresponding element of the row status array to SQL_ROW_ERROR for errors or SQL_ROW_SUCCESS_WITH_INFO for warnings.
- Adds zero or more status records containing SQLSTATEs for the error or warning.
- Sets the row and column number fields in the status records. If SQLFetch() cannot determine a row or column number, it sets that number to SQL_ROW_NUMBER_UNKNOWN or SQL_COLUMN_NUMBER_UNKNOWN respectively. If the status record does not apply to a particular column, SQLFetch() sets the column number to SQL_NO_COLUMN_NUMBER.

SQLFetch() continues fetching rows until it has fetched all of the rows in the rowset. It returns SQL_SUCCESS_WITH_INFO unless an error occurs in every row of the rowset (not counting rows with status SQL_ROW_NOROW), in which case it returns SQL_ERROR. In particular, if the rowset size is 1 and an error occurs in that row, SQLFetch() returns SQL_ERROR.

SQLFetch() returns the status records in row number order. That is, it returns all status records for unknown rows (if any), then all status records for the first row (if any), then all status records for the second row (if any), and so on. The status records for each individual row are ordered according to the normal rules for ordering status records; for more information, see SQLGetDiagField(), "Sequence of Status Records" on page 474.

Descriptors and SQLFetch

The following sections describe how SQLFetch() interacts with descriptors.

Argument Mappings

The driver does not set any descriptor fields based on the arguments of SQLFetch().

Other Descriptor Fields

The following descriptor fields are used by SQLFetch():

Table 68. Descriptor Fields

Descriptor field	Desc.	Location	Set through
SQL_DESC_ARRAY_SIZE	ARD	header	SQL_ATTR_ROW_ARRAY_SIZE statement attribute
SQL_DESC_ARRAY_STATUS_PTR	IRD	header	SQL_ATTR_ROW_STATUS_PTR statement attribute

Table 68. Descriptor Fields (continued)

Descriptor field	Desc.	Location	Set through
SQL_DESC_BIND_OFFSET_PTR	ARD	header	SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute
SQL_DESC_BIND_TYPE	ARD	header	SQL_ATTR_ROW_BIND_TYPE statement attribute
SQL_DESC_COUNT	ARD	header	*ColumnNumber* argument of SQLBindCol()
SQL_DESC_DATA_PTR	ARD	records	*TargetValuePtr* argument of SQLBindCol()
SQL_DESC_INDICATOR_PTR	ARD	records	*StrLen_or_IndPtr* argument in SQLBindCol()
SQL_DESC_OCTET_LENGTH	ARD	records	*BufferLength* argument in SQLBindCol()
SQL_DESC_OCTET_LENGTH_PTR	ARD	records	*StrLen_or_IndPtr* argument in SQLBindCol()
SQL_DESC_ROWS_PROCESSED_PTR	IRD	header	SQL_ATTR_ROWS_FETCHED_PTR statement attribute
SQL_DESC_TYPE	ARD	records	*TargetType* argument in SQLBindCol()

All descriptor fields can also be set through SQLSetDescField().

Separate Length and Indicator Buffers

Applications can bind a single buffer or two separate buffers to be used to hold length and indicator values. When an application calls SQLBindCol(), SQL_DESC_OCTET_LENGTH_PTR and SQL_DESC_INDICATOR_PTR fields of the ARD are set to the same address, which is passed in the *StrLen_or_IndPtr* argument. When an application calls SQLSetDescField() or SQLSetDescRec(), it can set these two fields to different addresses.

SQLFetch() determines whether the application has specified separate l ength and indicator buffers. In this case, when the data is not NULL, SQLFetch() sets the indicator buffer to 0 and returns the length in the length buffer. When the data is NULL, SQLFetch() sets the indicator buffer to SQL_NULL_DATA and does not modify the length buffer.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND is returned if there are no rows in the result set, or previous SQLFetch() calls have fetched all the rows from the result set.

If all the rows have been fetched, the cursor is positioned after the end of the result set.

Diagnostics

Table 69. SQLFetch SQLSTATEs

SQLSTATE	Description	Explanation
07009	Invalid descriptor index	Column 0 was bound but bookmarks are not being used (the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF).
01004	Data truncated.	The data returned for one or more columns was truncated. String values or numeric values are right truncated. (SQL_SUCCESS_WITH_INFO is returned if no error occurred.)
07002	Too many columns.	A column number specified in the binding for one or more columns was greater than the number of columns in the result set.
07006	Invalid conversion.	The data value could not be converted in a meaningful manner to the data type specified by *fCType* in SQLBindCol()
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument *pcbValue* in SQLBindCol() was a null pointer and the value of the corresponding column is null. There is no means to report SQL_NULL_DATA. The pointer specified for the argument IndicatorValue in SQLBindFileToCol() was a null pointer and the value of the corresponding LOB column is NULL. There is no means to report SQL_NULL_DATA.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for one or more columns would have caused the whole part of the number to be truncated either at the time of assignment or in computing an intermediate result. A value from an arithmetic expression was returned which resulted in division by zero. **Note:** The associated cursor is undefined if this error is detected by DB2 Universal Database. If the error was detected by DB2 CLI or by other IBM RDBMSs, the cursor will remain open and continue to advance on subsequent fetch calls.

Table 69. SQLFetch SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22005	Error in assignment.	A returned value was incompatible with the data type of binding. A returned LOB locator was incompatible with the data type of the bound column.
22007	Invalid datetime format.	Conversion from character a string to a datetime format was indicated, but an invalid string representation or value was specified, or the value was an invalid date. The value of a date, time, or timestamp does not conform to the syntax for the specified data type.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	A value from an arithmetic expression was returned which resulted in division by zero.
24000	Invalid cursor state.	The previous SQL statement executed on the statement handle was not a query.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.

Table 69. SQLFetch SQLSTATEs (continued)

SQLSTATE	Description	Explanation

428A1	Unable to access a file referenced by a host file variable.	This can be raised for any of the following scenarios. The associated reason code in the text identifies the particular error:
		• 01 - The file name length is invalid or the file name and/or the path has an invalid format.
		• 02 - The file option is invalid. It must have one of the following values:
		`SQL_FILE_READ -read from an existing file`
		`SQL_FILE_CREATE -create a new file for write`
		`SQL_FILE_OVERWRITE -overwrite an existing file. If the file does not exist, create the file.`
		`SQL_FILE_APPEND -append to an existing file. If the file does not exist, create the file.`
		• 03 - The file cannot be found.
		• 04 - The SQL_FILE_CREATE option was specified for a file with the same name as an existing file.
		• 05 - Access to the file was denied. The user does not have permission to open the file.
		• 06 - Access to the file was denied. The file is in use with incompatible modes. Files to be written to are opened in exclusive mode.
		• 07 - Disk full was encountered while writing to the file.
		• 08 - Unexpected end of file encountered while reading from the file.
		• 09 - A media error was encountered while accessing the file.

54028	The maximum number of concurrent LOB handles has been reached.	Maximum LOB locator assigned. The maximum number of concurrent LOB locators has been reached. A new locator can not be assigned.
56084	LOB data is not supported in DRDA.	LOBs not supported on DRDA. LOB columns cannot either be selected or updated when connecting to DRDA servers (using DB2 Connect).
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.

Table 69. SQLFetch SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, SQLCancel() was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, SQLCancel() was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	SQLFetch() was called for an StatementHandle after SQLExtendedFetch() was called and before SQLFreeStmt() had been called with the SQL_CLOSE option. The function was called prior to calling SQLPrepare() or SQLExecDirect() for the *StatementHandle*. The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY092	Option type out of range.	The *FileOptions* argument of a previous SQLBindFileToCol() operation was not valid.
HYC00	Driver not capable.	DB2 CLI or the data source does not support the conversion specified by the combination of the *fCType* in SQLBindCol() or SQLBindFileToCol() and the SQL data type of the corresponding column. A call to SQLBindCol() was made for a column data type which is not supported by DB2 CLI.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

None.

Example

```
/* From CLI sample fetch.c */
/* ... */
/******************************************************************
** main
******************************************************************/
int main( int argc, char * argv[] ) {

    SQLHANDLE henv, hdbc, hstmt ;
    SQLRETURN rc ;

    SQLCHAR * sqlstmt =
       "SELECT deptname, location from org where division = 'Eastern'" ;

    struct { SQLINTEGER ind ;
             SQLCHAR s[15] ;
           } deptname, location ;

/* ... */

    /* macro to initalize server, uid and pwd */
    INIT_UID_PWD ;

    /* allocate an environment handle */
    rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    /* allocate a connect handle, and connect */
    rc = DBconnect( henv, &hdbc ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

    rc = SQLExecDirect( hstmt, sqlstmt, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol( hstmt, 1, SQL_C_CHAR, deptname.s, 15, &deptname.ind ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol( hstmt, 2, SQL_C_CHAR, location.s, 15, &location.ind ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf( "Departments in Eastern division:\n" ) ;
    printf( "DEPTNAME       Location\n" ) ;
    printf( "-------------- ------------\n" ) ;

    while ( ( rc = SQLFetch( hstmt ) ) == SQL_SUCCESS )
       printf( "%-14.14s %-14.14s \n", deptname.s, location.s ) ;
    if ( rc != SQL_NO_DATA_FOUND )
       CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /*  Commit the changes.  */
    rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;
```

```
    /*  Disconnect and free up CLI resources.  */

    rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf( "\n>Disconnecting .....\n" ) ;
    rc = SQLDisconnect( hdbc ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

    rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

    rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    return( SQL_SUCCESS ) ;

}                                   /* end main */
```

References

- “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408
- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLBindFileToCol - Bind LOB File Reference to LOB Column” on page 237
- “SQLExecute - Execute a Statement” on page 373
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLGetData - Get Data From a Column” on page 447

SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLFetchScroll() fetches the specified rowset of data from the result set and returns data for all bound columns. Rowsets can be specified at an absolute or relative position or by bookmark.

Syntax

```
SQLRETURN   SQLFetchScroll   (SQLHSTMT          StatementHandle,
                              SQLSMALLINT       FetchOrientation,
                              SQLINTEGER        FetchOffset);
```

Function Arguments

Table 70. SQLFetchScroll Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLUSMALLINT	FetchOrientation	input	Type of fetch: • SQL_FETCH_NEXT • SQL_FETCH_PRIOR • SQL_FETCH_FIRST • SQL_FETCH_LAST • SQL_FETCH_ABSOLUTE • SQL_FETCH_RELATIVE • SQL_FETCH_BOOKMARK For more information, see “Positioning the Cursor” on page 409.
SQLINTEGER	FetchOffset	input	Number of the row to fetch. The interpretation of this argument depends on the value of the *FetchOrientation* argument. For more information, see “Positioning the Cursor” on page 409.

Usage

Overview

SQLFetchScroll() returns a specified rowset from the result set. Rowsets can be specified by absolute or relative position or by bookmark.
SQLFetchScroll() can be called only while a result set exists-that is, after a call that creates a result set and before the cursor over that result set is closed. If any columns are bound, it returns the data in those columns. If the

application has specified a pointer to a row status array or a buffer in which to return the number of rows fetched, SQLFetchScroll() returns this information as well. Calls to SQLFetchScroll() can be mixed with calls to SQLFetch() but cannot be mixed with calls to SQLExtendedFetch().

Positioning the Cursor

When the result set is created, the cursor is positioned before the start of the result set. SQLFetchScroll() positions the block cursor based on the values of the *FetchOrientation* and *FetchOffset* arguments as shown in the following table. The exact rules for determining the start of the new rowset are shown in the next section.

FetchOrientation	Meaning
SQL_FETCH_NEXT	Return the next rowset. This is equivalent to calling SQLFetch(). SQLFetchScroll() ignores the value of *FetchOffset*.
SQL_FETCH_PRIOR	Return the prior rowset. SQLFetchScroll() ignores the value of *FetchOffset*.
SQL_FETCH_RELATIVE	Return the rowset *FetchOffset* from the start of the current rowset.
SQL_FETCH_ABSOLUTE	Return the rowset starting at row *FetchOffset*.
SQL_FETCH_FIRST	Return the first rowset in the result set. SQLFetchScroll() ignores the value of *FetchOffset*.
SQL_FETCH_LAST	Return the last complete rowset in the result set. SQLFetchScroll() ignores the value of *FetchOffset*.
SQL_FETCH_BOOKMARK	Return the rowset *FetchOffset* rows from the bookmark specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute.

The SQL_ATTR_ROW_ARRAY_SIZE statement attribute specifies the number of rows in the rowset. If the rowset being fetched by SQLFetchScroll() overlaps the end of the result set, SQLFetchScroll() returns a partial rowset. That is, if S + R-1 is greater than L, where S is the starting row of the rowset being fetched, R is the rowset size, and L is the last row in the result set, then only the first L-S+1 rows of the rowset are valid. The remaining rows are empty and have a status of SQL_ROW_NOROW.

After SQLFetchScroll() returns, the rowset cursor is positioned on the first row of the result set.

Cursor Positioning Rules

The following sections describe the exact rules for each value of *FetchOrientation*. These rules use the following notation:

FetchOrientation
Meaning

Before start The block cursor is positioned before the start of the result set. If the first row of the new rowset is before the start of the result set, SQLFetchScroll() returns SQL_NO_DATA.

After end The block cursor is positioned after the end of the result set. If the first row of the new rowset is after the end of the result set, SQLFetchScroll() returns SQL_NO_DATA.

CurrRowsetStart
The number of the first row in the current rowset.

LastResultRow
The number of the last row in the result set.

RowsetSize The rowset size.

FetchOffset The value of the *FetchOffset* argument.

BookmarkRow
The row corresponding to the bookmark specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute.

SQL_FETCH_NEXT rules:

Table 71. SQL_FETCH_NEXT Rules:

Condition	First row of new rowset
Before start	1
CurrRowsetStart + RowsetSize <= LastResultRow	CurrRowsetStart + RowsetSize
CurrRowsetStart + RowsetSize > LastResultRow	After end
After end	After end

SQL_FETCH_PRIOR rules:

Table 72. SQL_FETCH_PRIOR Rules:

Condition	First row of new rowset
Before start	Before start
CurrRowsetStart = 1	Before start
1 < CurrRowsetStart <= RowsetSize	1 [a]

Table 72. SQL_FETCH_PRIOR Rules: (continued)

Condition	First row of new rowset
CurrRowsetStart > RowsetSize	CurrRowsetStart - RowsetSize
After end AND LastResultRow < RowsetSize	1 [a]
After end AND LastResultRow >= RowsetSize	LastResultRow - RowsetSize + 1

a `SQLFetchScroll()` returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first rowset.) and SQL_SUCCESS_WITH_INFO.

SQL_FETCH_RELATIVE rules:

Table 73. SQL_FETCH_RELATIVE Rules:

Condition	First row of new rowset
(Before start AND *FetchOffset* > 0) OR (After end AND *FetchOffset* < 0)	-- [a]
Before start AND *FetchOffset* <= 0	Before start
CurrRowsetStart = 1 AND FetchOffset < 0	Before start
CurrRowsetStart > 1 AND CurrRowsetStart + *FetchOffset* < 1 AND |*FetchOffset*| > RowsetSize	Before start
CurrRowsetStart > 1 AND CurrRowsetStart + *FetchOffset* < 1 AND |*FetchOffset*| <= RowsetSize	1 [b]
1 <= CurrRowsetStart + *FetchOffset* <= LastResultRow	CurrRowsetStart + *FetchOffset*
CurrRowsetStart + *FetchOffset* > LastResultRow	After end
After end AND *FetchOffset* >= 0	After end

a `SQLFetchScroll()` returns the same rowset as if it was called with *FetchOrientation* set to SQL_FETCH_ABSOLUTE. For more information, see the "SQL_FETCH_ABSOLUTE" section.

b `SQLFetchScroll()` returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first rowset.) and SQL_SUCCESS_WITH_INFO.

SQL_FETCH_ABSOLUTE rules:

Table 74. SQL_FETCH_ABSOLUTE Rules:

Condition	First row of new rowset
FetchOffset < 0 AND |*FetchOffset*| <= LastResultRow	LastResultRow + *FetchOffset* + 1
FetchOffset < 0 AND |*FetchOffset*| > LastResultRow AND |FetchOffset| > RowsetSize	Before start

Table 74. SQL_FETCH_ABSOLUTE Rules: (continued)

Condition	First row of new rowset
FetchOffset < 0 AND |*FetchOffset*| > LastResultRow AND |FetchOffset| <= RowsetSize	1 [a]
FetchOffset = 0	Before start
1 <= *FetchOffset* <= LastResultRow	*FetchOffset*
FetchOffset > LastResultRow	After end

a SQLFetchScroll() returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first rowset.) and SQL_SUCCESS_WITH_INFO.

SQL_FETCH_FIRST rules:

Table 75. SQL_FETCH_FIRST Rules:

Condition	First row of new rowset
Any	1

SQL_FETCH_LAST rules:

Table 76. SQL_FETCH_LAST Rules:

Condition	First row of new rowset
RowsetSize <= LastResultRow	LastResultRow - RowsetSize + 1
RowsetSize > LastResultRow	1

SQL_FETCH_BOOKMARK rules:

Table 77. SQL_FETCH_BOOKMARK Rules:

Condition	First row of new rowset
BookmarkRow + *FetchOffset* < 1	Before start
1 <= BookmarkRow + *FetchOffset* <= LastResultRow	BookmarkRow +*FetchOffset*
BookmarkRow + *FetchOffset* > LastResultRow	After end

Returning Data in Bound Columns

SQLFetchScroll() returns data in bound columns in the same way as SQLFetch(). For more information see "SQLFetch - Fetch Next Row" on page 396.

If no columns are bound, SQLFetchScroll() does not return data but does move the block cursor to the specified position. As with SQLFetch(), you can use SQLGetData() to retrieve the information in this case.

Buffer Addresses

SQLFetchScroll() uses the same formula to determine the address of data and length/indicator buffers as SQLFetch(). For more information, see "Buffer Addresses" in SQLBindCol().

Row Status Array

The row status array is used to return the status of each row in the rowset. The address of this array is specified with the SQL_ATTR_ROW_STATUS_PTR statement attribute. The array is allocated by the application and must have as many elements as are specified by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Its values are set by SQLFetch(), SQLFetchScroll() SQLSetPos() (except when they have been called after the cursor has been positioned by SQLExtendedFetch()). If the value of the SQL_ATTR_ROW_STATUS_PTR statement attribute is a null pointer, these functions do not return the row status.

The contents of the row status array buffer are undefined if SQLFetch() or SQLFetchScroll() does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

The following values are returned in the row status array.

Row status array value	Description
SQL_ROW_SUCCESS	The row was successfully fetched.
SQL_ROW_SUCCESS_WITH_INFO	The row was successfully fetched. However, a warning was returned about the row.
SQL_ROW_ERROR	An error occurred while fetching the row.
SQL_ROW_ADDED	The row was inserted by SQLBulkOperations(). If the row is fetched again, or is refreshed by SQLSetPos() its status is SQL_ROW_SUCCESS. This value is not set by SQLFetch() or SQLFetchScroll().
SQL_ROW_UPDATED	The row was successfully fetched and has changed since it was last fetched from this result set. If the row is fetched again from this result set, or is refreshed by SQLSetPos(), the status changes to the row's new status.
SQL_ROW_DELETED	The row has been deleted since it was last fetched from this result set.

SQL_ROW_NOROW	The rowset overlapped the end of the result set and no row was returned that corresponded to this element of the row status array.

Rows Fetched Buffer

The rows fetched buffer is used to return the number of rows fetched, including those rows for which no data was returned because an error occurred while they were being fetched. In other words, it is the number of rows for which the value in the row status array is not SQL_ROW_NOROW. The address of this buffer is specified with the SQL_ATTR_ROWS_FETCHED_PTR statement attribute. The buffer is allocated by the application. It is set by `SQLFetch()` and `SQLFetchScroll()`. If the value of the SQL_ATTR_ROWS_FETCHED_PTR statement attribute is a null pointer, these functions do not return the number of rows fetched. To determine the number of the current row in the result set, an application can call `SQLGetStmtAttr()` with the SQL_ATTR_ROW_NUMBER attribute.

The contents of the rows fetched buffer are undefined if `SQLFetch()` or `SQLFetchScroll()` does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, except when SQL_NO_DATA is returned, in which case the value in the rows fetched buffer is set to 0.

Error Handling

`SQLFetchScroll()` returns errors and warnings in the same manner as `SQLFetch()`; for more information see `SQLFetch()`, "Error Handling" on page 399. `SQLExtendedFetch()` returns errors in the same manner as `SQLFetch()` with the following exceptions:

- When a warning occurs that applies to a particular row in the rowset, `SQLExtendedFetch()` sets the corresponding entry in the row status array to SQL_ROW_SUCCESS, not SQL_ROW_SUCCESS_WITH_INFO.
- If errors occur in every row in the rowset, `SQLExtendedFetch()` returns SQL_SUCCESS_WITH_INFO, not SQL_ERROR.
- In each group of status records that applies to an individual row, the first status record returned by `SQLExtendedFetch()` must contain SQLSTATE 01S01 (Error in row); `SQLFetchScroll()` does not return this SQLSTATE. Note that if `SQLExtendedFetch()` is unable to return additional SQLSTATEs, it still must return this SQLSTATE.

Descriptors and SQLFetchScroll()

SQLFetchScroll() interacts with descriptors in the same manner as SQLFetch(). For more information see SQLFetch(), "Descriptors and SQLFetch" on page 400.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise. If an error occurs on a single column, SQLGetDiagField() can be called with a *DiagIdentifier* of SQL_DIAG_COLUMN_NUMBER to determine the column the error occurred on; and SQLGetDiagField() can be called with a *DiagIdentifier* of SQL_DIAG_ROW_NUMBER to determine the row containing that column.

Table 78. SQLFetchScroll SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	String or binary data returned for a column resulted in the truncation of non-blank character or non-NULL binary data. String values are right truncated. (Function returns SQL_SUCCESS_WITH_INFO.)
01S01	Error in row.	An error occurred while fetching one or more rows. (Function returns SQL_SUCCESS_WITH_INFO.) (This SQLSTATE is only returned when connected to DB2 CLI v2.)
01S06	Attempt to fetch before the result set returned the first rowset.	The requested rowset overlapped the start of the result set when the current position was beyond the first row, and either *FetchOrientation* was SQL_PRIOR, or *FetchOrientation* was SQL_RELATIVE with a negative *FetchOffset* whose absolute value was less than or equal to the current SQL_ATTR_ROW_ARRAY_SIZE. (Function returns SQL_SUCCESS_WITH_INFO.)

SQLFetchScroll

Table 78. SQLFetchScroll SQLSTATEs (continued)

SQLSTATE	Description	Explanation
01S07	Fractional truncation.	The data returned for a column was truncated. For numeric data types, the fractional part of the number was truncated. For time, timestamp, and interval data types containing a time component, the fractional portion of the time was truncated.
07002	Too many columns.	A column number specified in the binding for one or more columns was greater than the number of columns in the result set.
07006	Invalid conversion.	A data value of a column in the result set could not be converted to the C data type specified by *TargetType* in `SQLBindCol()`.
07009	Invalid descriptor index.	Column 0 was bound and the SQL_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was connected failed before the function completed processing.
22001	String data right truncation.	A variable-length bookmark returned for a row was truncated.
22002	Invalid output or indicator buffer specified.	NULL data was fetched into a column whose *StrLen_or_IndPtr* set by `SQLBindCol()` (or SQL_DESC_INDICATOR_PTR set by `SQLSetDescField()` or `SQLSetDescRec()`) was a null pointer.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for one or more bound columns would have caused the whole (as opposed to fractional) part of the number to be truncated.
22007	Invalid datetime format.	A character column in the result set was bound to a date, time, or timestamp C structure, and a value in the column was, respectively, an invalid date, time, or timestamp.
22012	Division by zero is invalid.	A value from an arithmetic expression was returned which resulted in division by zero.
22018	Invalid character value for cast specification.	A character column in the result set was bound to a character C buffer and the column contained a character for which there was no representation in the character set of the buffer. A character column in the result set was bound to an approximate numeric C buffer and a value in the column could not be cast to a valid approximate numeric value. A character column in the result set was bound to an exact numeric C buffer and a value in the column could not be cast to a valid exact numeric value. A character column in the result set was bound to a datetime or interval C buffer and a value in the column could not be cast to a valid datetime or interval value.

Table 78. SQLFetchScroll SQLSTATEs (continued)

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	The *StatementHandle* was in an executed state but no result set was associated with the *StatementHandle*.
40001	Transaction rollback.	The transaction in which the fetch was executed was terminated to prevent deadlock.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY008	Operation was cancelled	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The specified *StatementHandle* was not in an executed state. The function was called without first calling `SQLExecDirect()`, `SQLExecute()`, or a catalog function. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()` or `SQLExecDirect()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. `SQLFetchScroll()` was called for the *StatementHandle* after `SQLFetch()` was called, and was connected to a DB2 v2 or earlier server, and either before `SQLFreeStmt()` was called with the SQL_CLOSE option, or before `SQLMoreResults()` was called. `SQLFetchScroll()` was called for a *StatementHandle* after `SQLExtendedFetch()` was called and before `SQLFreeStmt()` with SQL_CLOSE was called.

Table 78. SQLFetchScroll SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY106	Fetch type out of range.	The value specified for the argument *FetchOrientation* was invalid. The argument *FetchOrientation* was SQL_FETCH_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF. The value of the SQL_CURSOR_TYPE statement attribute was SQL_CURSOR_FORWARD_ONLY and the value of argument *FetchOrientation* was not SQL_FETCH_NEXT.
HY107	Row value out of range.	The value specified with the SQL_ATTR_CURSOR_TYPE statement attribute was SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the SQL_ATTR_KEYSET_SIZE statement attribute was greater than 0 and less than the value specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.
HY111	Invalid bookmark value.	The argument *FetchOrientation* was SQL_FETCH_BOOKMARK and the bookmark pointed to by the value in the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute was not valid or was a null pointer.
HYC00	Driver not capable.	The specified fetch type is not supported. The conversion specified by the combination of the *TargetType* in `SQLBindCol()` and the SQL data type of the corresponding column is not supported.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLCancel - Cancel Statement” on page 290
- “SQLDescribeCol - Return a Set of Attributes for a Column” on page 336
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLExecute - Execute a Statement” on page 373

- “SQLFetch - Fetch Next Row” on page 396
- “SQLNumResultCols - Get Number of Result Columns” on page 572
- “SQLSetPos - Set the Cursor Position in a Rowset” on page 691
- “SQLSetStmtAttr - Set Options Related to a Statement” on page 702

SQLForeignKeys - Get the List of Foreign Key Columns

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLForeignKeys() returns information about foreign keys for the specified table. The information is returned in an SQL result set which can be processed using the same functions that are used to retrieve a result generated by a query.

Syntax

```
SQLRETURN   SQLForeignKeys   (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLCHAR      *FAR PKCatalogName,     /* szPkCatalogName */
                SQLSMALLINT       NameLength1,       /* cbPkCatalogName */
                SQLCHAR      *FAR PKSchemaName,      /* szPkSchemaName */
                SQLSMALLINT       NameLength2,       /* cbPkSchemaName */
                SQLCHAR      *FAR PKTableName,       /* szPkTableName */
                SQLSMALLINT       NameLength3,       /* cbPkTableName */
                SQLCHAR      *FAR FKCatalogName,     /* szFkCatalogName */
                SQLSMALLINT       NameLength4,       /* cbFkCatalogName */
                SQLCHAR      *FAR FKSchemaName,      /* szFkSchemaName */
                SQLSMALLINT       NameLength5,       /* cbFkSchemaName */
                SQLCHAR      *FAR FKTableName,       /* szFkTableName */
                SQLSMALLINT       NameLength6);      /* cbFkTableName */
```

Function Arguments

Table 79. SQLForeignKeys Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLCHAR *	PKCatalogName	input	Catalog qualifier of the primary key table. This must be a NULL pointer or a zero length string.
SQLSMALLINT	NameLength1	input	Length of *PKCatalogName*. This must be set to 0.
SQLCHAR *	PKSchemaName	input	Schema qualifier of the primary key table.
SQLSMALLINT	NameLength2	input	Length of *PKSchemaName*
SQLCHAR *	PKTableName	input	Name of the table name containing the primary key.
SQLSMALLINT	NameLength3	input	Length of *PKTableName*
SQLCHAR *	FKCatalogName	input	Catalog qualifier of the table containing the foreign key. This must be a NULL pointer or a zero length string.

Table 79. SQLForeignKeys Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	NameLength4	input	Length of *FKCatalogName*. This must be set to 0.
SQLCHAR *	FKSchemaName	input	Schema qualifier of the table containing the foreign key.
SQLSMALLINT	NameLength5	input	Length of *FKSchemaName*
SQLCHAR *	FKTableName	input	Name of the table containing the foreign key.
SQLSMALLINT	NameLength6	input	Length of *FKTableName*

Usage

If *PKTableName* contains a table name, and *FKTableName* is an empty string, `SQLForeignKeys()` returns a result set containing the primary key of the specified table and all of the foreign keys (in other tables) that refer to it.

If *FKTableName* contains a table name, and *PKTableName* is an empty string, `SQLForeignKeys()` returns a result set containing all of the foreign keys in the specified table and the primary keys (in other tables) to which they refer.

If both *PKTableName* and *FKTableName* contain table names, `SQLForeignKeys()` returns the foreign keys in the table specified in *FKTableName* that refer to the primary key of the table specified in *PKTableName*. This should be one key at the most.

If the schema qualifier argument associated with a table name is not specified, then the schema name defaults to the one currently in effect for the current connection.

Table 80 on page 422 lists the columns of the result set generated by the `SQLForeignKeys()` call. If the foreign keys associated with a primary key are requested, the result set is ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and ORDINAL_POSITION. If the primary keys associated with a foreign key are requested, the result set is ordered by PKTABLE_CAT, PKTABLE_SCHEM, PKTABLE_NAME, and ORDINAL_POSITION.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual

lengths of the associated TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Table 80. Columns Returned By SQLForeignKeys

Column Number/Name	Data Type	Description
1 PKTABLE_CAT	VARCHAR(128)	This is always NULL.
2 PKTABLE_SCHEM	VARCHAR(128)	The name of the schema containing PKTABLE_NAME.
3 PKTABLE_NAME	VARCHAR(128) not NULL	Name of the table containing the primary key.
4 PKCOLUMN_NAME	VARCHAR(128) not NULL	Primary key column name.
5 FKTABLE_CAT	VARCHAR(128)	This is always NULL.
6 FKTABLE_SCHEM	VARCHAR(128)	The name of the schema containing FKTABLE_NAME.
7 FKTABLE_NAME	VARCHAR(128) not NULL	The name of the table containing the Foreign key.
8 FKCOLUMN_NAME	VARCHAR(128) not NULL	Foreign key column name.
9 ORDINAL_POSITION	SMALLINT not NULL	The ordinal position of the column in the key, starting at 1.
10 UPDATE_RULE	SMALLINT	Action to be applied to the foreign key when the SQL operation is UPDATE: • SQL_RESTRICT • SQL_NO_ACTION The update rule for IBM DB2 DBMSs is always either RESTRICT or SQL_NO_ACTION. However, ODBC applications may encounter the following UPDATE_RULE values when connected to non-IBM RDBMSs: • SQL_CASCADE • SQL_SET_NULL

Table 80. Columns Returned By SQLForeignKeys (continued)

Column Number/Name	Data Type	Description
11 DELETE_RULE	SMALLINT	Action to be applied to the foreign key when the SQL operation is DELETE: • SQL_CASCADE • SQL_NO_ACTION • SQL_RESTRICT • SQL_SET_DEFAULT • SQL_SET_NULL
12 FK_NAME	VARCHAR(128)	Foreign key identifier. NULL if not applicable to the data source.
13 PK_NAME	VARCHAR(128)	Primary key identifier. NULL if not applicable to the data source.
14 DEFERRABILITY	SMALLINT	One of: • SQL_INITIALLY_DEFERRED • SQL_INITIALLY_IMMEDIATE • SQL_NOT_DEFERRABLE

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLForeignKeys()` result set in ODBC.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 81. SQLForeignKeys SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY009	Invalid argument value.	The arguments *PKTableName* and *FKTableName* were both NULL pointers.

Table 81. SQLForeignKeys SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal SQL_NTS. The length of the table or owner name is greater than the maximum length supported by the server. Refer to "SQLGetInfo - Get General Information" on page 489.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

None.

Example

```
/* From CLI sample browser.c */
/* ... */
SQLRETURN list_foreign_keys( SQLHANDLE hstmt,
                             SQLCHAR * schema,
                             SQLCHAR * tablename
                           ) {

/* ... */
    rc = SQLForeignKeys(hstmt, NULL, 0,
                        schema, SQL_NTS, tablename, SQL_NTS,
                        NULL, 0,
                        NULL, SQL_NTS, NULL, SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

```
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) pktable_schem.s, 129,
                &pktable_schem.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) pktable_name.s, 129,
                &pktable_name.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) pkcolumn_name.s, 129,
                &pkcolumn_name.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 6, SQL_C_CHAR, (SQLPOINTER) fktable_schem.s, 129,
                &fktable_schem.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) fktable_name.s, 129,
                &fktable_name.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 8, SQL_C_CHAR, (SQLPOINTER) fkcolumn_name.s, 129,
                &fkcolumn_name.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 10, SQL_C_SHORT, (SQLPOINTER) &update_rule,
                0, &update_ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 11, SQL_C_SHORT, (SQLPOINTER) &delete_rule,
                0, &delete_ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 12, SQL_C_CHAR, (SQLPOINTER) fkey_name.s, 129,
                &fkey_name.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 13, SQL_C_CHAR, (SQLPOINTER) pkey_name.s, 129,
                &pkey_name.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

printf("Primary Key and Foreign Keys for %s.%s\n", schema, tablename);
/* Fetch each row, and display */
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
    printf("  %s  %s.%s.%s\n      Update Rule ",
           pkcolumn_name.s, fktable_schem.s,
           fktable_name.s, fkcolumn_name.s);
    if (update_rule == SQL_RESTRICT) {
        printf("RESTRICT ");  /* always for IBM DBMSs */
    } else {
        if (update_rule == SQL_CASCADE) {
           printf("CASCADE ");  /* non-IBM only */
        } else {
           printf("SET NULL ");
        }
    }
```

```
        printf(", Delete Rule: ");
        if (delete_rule== SQL_RESTRICT) {
            printf("RESTRICT ");  /* always for IBM DBMSs */
        } else {
            if (delete_rule == SQL_CASCADE) {
               printf("CASCADE ");  /* non-IBM only */
            } else {
               if (delete_rule == SQL_NO_ACTION) {
                  printf("NO ACTION ");  /* non-IBM only */
               } else {
                    printf("SET NULL ");
                 }
            }
        }
        printf("\n");
        if (pkey_name.ind > 0 ) {
            printf("     Primary Key Name: %s\n", pkey_name.s);
        }
        if (fkey_name.ind > 0 ) {
            printf("     Foreign Key Name: %s\n", fkey_name.s);
        }
    }
```

References

- “SQLPrimaryKeys - Get Primary Key Columns of A Table” on page 590
- “SQLStatistics - Get Index and Statistics Information For A Base Table” on page 733

SQLFreeConnect - Free Connection Handle

Deprecated

Note:

In ODBC version 3, SQLFreeConnect() has been deprecated and replaced with SQLFreeHandle(); see "SQLFreeHandle - Free Handle Resources" on page 431 for more information.

Although this version of DB2 CLI continues to support SQLFreeConnect(), we recommend that you begin using SQLFreeHandle() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Migrating to the New Function

The statement:

```
SQLFreeConnect(hdbc);
```

for example, would be rewritten using the new function as:

```
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
```

Purpose

Specification:	**DB2 CLI** 1.1	**ODBC** 1.0	**ISO CLI**

SQLFreeConnect() invalidates and frees the connection handle. All DB2 CLI resources associated with the connection handle are freed.

SQLDisconnect() must be called before calling this function.

Syntax

```
SQLRETURN   SQLFreeConnect   (SQLHDBC          hdbc);
```

Function Arguments

Table 82. SQLFreeConnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle

SQLFreeConnect

Usage

If this function is called when a connection still exists, SQL_ERROR is returned, and the connection handle remains valid.

To continue termination, call SQLFreeEnv(), or, if a new connection handle is required, call SQLAllocConnect().

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 83. SQLFreeConnect SQLSTATEs

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function was called prior to SQLDisconnect() for the *hdbc*.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to "SQLFreeHandle - Free Handle Resources" on page 431.

References

- "SQLDisconnect - Disconnect from a Data Source" on page 347
- "SQLFreeHandle - Free Handle Resources" on page 431

SQLFreeEnv - Free Environment Handle

Deprecated

Note:

In ODBC version 3, SQLFreeEnv() has been deprecated and replaced with SQLFreeHandle(); see "SQLFreeHandle - Free Handle Resources" on page 431 for more information.

Although this version of DB2 CLI continues to support SQLFreeEnv(), we recommend that you begin using SQLFreeHandle() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Migrating to the New Function

The statement:

```
SQLFreeEnv(henv);
```

for example, would be rewritten using the new function as:

```
SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

Purpose

Specification:	**DB2 CLI** 1.1	**ODBC** 1.0	**ISO CLI**

SQLFreeEnv() invalidates and frees the environment handle. All DB2 CLI resources associated with the environment handle are freed.

SQLFreeConnect() must be called before calling this function.

This function is the last DB2 CLI step an application needs to do before terminating.

Syntax

```
SQLRETURN   SQLFreeEnv      (SQLHENV          henv);
```

Function Arguments

Table 84. SQLFreeEnv Arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle

SQLFreeEnv

Usage

If this function is called when there is still a valid connection handle, SQL_ERROR is returned, and the environment handle will remain valid.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 85. SQLFreeEnv SQLSTATEs

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	There is an *hdbc* which is in allocated or connected state. Call `SQLDisconnect()` and `SQLFreeConnect()` for the *hdbc* before calling `SQLFreeEnv()`.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Authorization

None.

Example

Refer to “SQLFreeHandle - Free Handle Resources” on page 431.

References

- “SQLFreeHandle - Free Handle Resources” on page 431

SQLFreeHandle - Free Handle Resources

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLFreeHandle() frees resources associated with a specific environment, connection, statement, or descriptor handle.

Note: This function is a generic function for freeing resources. It replaces the Version 2 functions SQLFreeConnect (for freeing a connection handle), and SQLFreeEnv() (for freeing an environment handle). SQLFreeHandle() also replaces the Version 2 function SQLFreeStmt() (with the SQL_DROP Option) for freeing a statement handle.

Syntax

```
SQLRETURN   SQLFreeHandle    (SQLSMALLINT       HandleType,
                              SQLHANDLE         Handle);
```

Function Arguments

Table 86. SQLFreeHandle Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	The type of handle to be freed by SQLFreeHandle(). Must be one of the following values: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC If *HandleType* is not one of the above values, SQLFreeHandle() returns SQL_INVALID_HANDLE.
SQLHANDLE	*Handle*	input	The handle to be freed.

Usage

SQLFreeHandle() is used to free handles for environments, connections, statements, and descriptors, as described below.

An application should not use a handle after it has been freed; DB2 CLI does not check the validity of a handle in a function call.

Freeing an Environment Handle

Prior to calling SQLFreeHandle() with a *HandleType* of SQL_HANDLE_ENV, an application must call SQLFreeHandle() with a *HandleType* of SQL_HANDLE_DBC for all connections allocated under the environment. Otherwise, the call to SQLFreeHandle() returns SQL_ERROR and the environment and any active connection remains valid.

Freeing a Connection Handle

Prior to calling SQLFreeHandle() with a *HandleType* of SQL_HANDLE_DBC, an application must call SQLDisconnect() for the connection. Otherwise, the call to SQLFreeHandle() returns SQL_ERROR and the connection remains valid.

Freeing a Statement Handle

A call to SQLFreeHandle() with a *HandleType* of SQL_HANDLE_STMT frees all resources that were allocated by a call to SQLAllocHandle() with a *HandleType* of SQL_HANDLE_STMT. When an application calls SQLFreeHandle() to free a statement that has pending results, the pending results are deleted. When an application frees a statement handle, DB2 CLI frees all the automatically generated descriptors associated with that handle. If there are results pending when SQLFreeHandle() is called, the result sets are discarded.

Note that SQLDisconnect() automatically drops any statements and descriptors open on the connection.

Freeing a Descriptor Handle

A call to SQLFreeHandle() with a *HandleType* of SQL_HANDLE_DESC frees the descriptor handle in *Handle*. The call to SQLFreeHandle() does not release any memory allocated by the application that may be referenced by the deferred fields (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR) of any descriptor record of *Handle*. When an explicitly allocated descriptor handle is freed, all statements that the freed handle had been associated with revert to their automatically allocated descriptor handle.

Note that SQLDisconnect() automatically drops any statements and descriptors open on the connection. When an application frees a statement handle, DB2 CLI frees all the automatically generated descriptors associated with that handle.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

If SQLFreeHandle() returns SQL_ERROR, the handle is still valid.

Diagnostics

Table 87. SQLFreeHandle SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure.	The *HandleType* argument was SQL_HANDLE_DBC, and the communication link between DB2 CLI and the data source to which it was trying to connect failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by SQLGetDiagRec() in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The *HandleType* argument was SQL_HANDLE_ENV, and at least one connection was in an allocated or connected state. SQLDisconnect() and SQLFreeHandle() with a *HandleType* of SQL_HANDLE_DBC must be called for each connection before calling SQLFreeHandle() with a *HandleType* of SQL_HANDLE_ENV. The *HandleType* argument was SQL_HANDLE_DBC, and the function was called before calling SQLDisconnect() for the connection. The *HandleType* argument was SQL_HANDLE_STMT; an asynchronously executing function was called on the statement handle; and the function was still executing when this function was called. The *HandleType* argument was SQL_HANDLE_STMT; SQLExecute() or SQLExecDirect() was called with the statement handle, and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) All subsidiary handles and other resources were not released before SQLFreeHandle() was called.
HY013	Unexpected memory handling error.	The *HandleType* argument was SQL_HANDLE_STMT or SQL_HANDLE_DESC, and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.

Table 87. SQLFreeHandle SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY017	Invalid use of an automatically allocated descriptor handle.	The *Handle* argument was set to the handle for an automatically allocated descriptor or an implementation descriptor.

Restrictions

None.

Example

See `SQLBrowseConnect()` and `SQLConnect()`.

References

- "SQLAllocHandle - Allocate Handle" on page 220
- "SQLCancel - Cancel Statement" on page 290
- "SQLSetCursorName - Set Cursor Name" on page 645

SQLFreeStmt - Free (or Reset) a Statement Handle

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLFreeStmt() ends processing on the statement referenced by the statement handle. Use this function to:

- Close a cursor
- Disassociate (reset) parameters from application variables and LOB file references
- Unbind columns from application variables and LOB file references
- To drop the statement handle and free the DB2 CLI resources associated with the statement handle.

SQLFreeStmt() is called after executing an SQL statement and processing the results.

Syntax

```
SQLRETURN   SQLFreeStmt      (SQLHSTMT          StatementHandle,   /* hstmt */
                              SQLUSMALLINT      Option);           /* fOption */
```

Function Arguments

Table 88. SQLFreeStmt Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLUSMALLINT	*Option*	input	Option which specified the manner of freeing the statement handle. The option must have one of the following values: • SQL_CLOSE • SQL_DROP • SQL_UNBIND • SQL_RESET_PARAMS

Usage

SQLFreeStmt() can be called with the following options:

SQL_CLOSE The cursor (if any) associated with the statement handle (*StatementHandle*) is closed and all pending results are discarded. The application can reopen the cursor by calling SQLExecute() with the same or different values in the application variables (if any) that are bound to *StatementHandle*. The cursor name is retained until the

statement handle is dropped or the next successful SQLSetCursorName() call. If no cursor has been associated with the statement handle, this option has no effect (no warning or error is generated).

SQLCloseCursor() can also be used to close a cursor.

SQL_DROP DB2 CLI resources associated with the input statement handle are freed, and the handle is invalidated. The open cursor, if any, is closed and all pending results are discarded.

This option has been replaced with a call to SQLFreeHandle() with the *HandleType* set to SQL_HANDLE_STMT. Although this version of DB2 CLI continues to support this option, we recommend that you begin using SQLFreeHandle() in your DB2 CLI programs so that they conform to the latest standards.

SQL_UNBIND

Sets the SQL_DESC_COUNT field of the ARD to 0, releasing all column buffers bound by SQLBindCol() or SQLBindFileToCol() for the given *StatementHandle*. This does not unbind the bookmark column; to do that, the SQL_DESC_DATA_PTR field of the ARD for the bookmark column is set to NULL. Note that if this operation is performed on an explicitly allocated descriptor that is shared by more than one statement, the operation will affect the bindings of all statements that share the descriptor.

SQL_RESET_PARAMS

Sets the SQL_DESC_COUNT field of the APD to 0, releasing all parameter buffers set by SQLBindParameter() or SQLBindFileToParam() for the given *StatementHandle*. Note that if this operation is performed on an explicitly allocated descriptor that is shared by more than one statement, this operation will affect the bindings of all the statements that share the descriptor.

SQLFreeStmt() has no effect on LOB locators, call SQLExecDirect() with the FREE LOCATOR statement to free a locator. Refer to “Using Large Objects” on page 116 for more information on using LOBs.

In order to reuse a statement handle to execute a different statement when the handle associated with a query, catalog function or SQLGetTypeInfo() was:

- Associated with a query, catalog function or SQLGetTypeInfo(), you must close the cursor.
- Bound with a different number or type of parameters, the parameters must be reset.

- Bound with a different number or type of column bindings, the columns must be unbound.

Alternatively you may drop the statement handle and allocate a new one.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

SQL_SUCCESS_WITH_INFO is not returned if *Option* is set to SQL_DROP, since there would be no statement handle to use when `SQLError()` is called.

Diagnostics

Table 89. SQLFreeStmt SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
HY092	Option type out of range.	The value specified for the argument *Option* was not SQL_CLOSE, SQL_DROP, SQL_UNBIND, or SQL_RESET_PARAMS.
HY506	Error closing a file.	Error encountered while trying to close a temporary file.

Authorization

None.

Example

Refer to “Example” on page 406.

References

- “SQLAllocHandle - Allocate Handle” on page 220
- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227

SQLFreeStmt

- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247
- “SQLBindFileToCol - Bind LOB File Reference to LOB Column” on page 237
- “SQLBindFileToParam - Bind LOB File Reference to LOB Parameter” on page 242
- “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408
- “SQLFetch - Fetch Next Row” on page 396
- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247

SQLGetConnectAttr - Get Current Attribute Setting

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLGetConnectAttr() returns the current setting of a connection attribute.

Syntax

```
SQLRETURN   SQLGetConnectAttr(SQLHDBC           ConnectionHandle,
                              SQLINTEGER        Attribute,
                              SQLPOINTER        ValuePtr,
                              SQLINTEGER        BufferLength,
                              SQLINTEGER        *StringLengthPtr);
```

Function Arguments

Table 90. SQLGetConnectAttr Arguments

Data Type	Argument	Use	Description
SQLHDBC	ConnectionHandle	input	Connection handle.
SQLINTEGER	Attribute	input	*Attribute* to retrieve.
SQLPOINTER	ValuePtr	output	A pointer to memory in which to return the current value of the attribute specified by *Attribute*.
SQLINTEGER	BufferLength	input	• If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*. • If *ValuePtr* is a pointer, but not to a string, then *BufferLength* should have the value SQL_IS_POINTER. • If *ValuePtr* is not a pointer, then *BufferLength* should have the value SQL_IS_NOT_POINTER. • If the value in **ValuePtr* is a unicode string the *BufferLength* argument must be an even number.

Table 90. SQLGetConnectAttr Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER	*StringLengthPtr	output	A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character) available to return in **ValuePtr*. If *ValuePtr* is a null pointer, no length is returned. If the attribute value is a character string, and the number of bytes available to return is greater than BufferLength minus the length of the null-termination character, the data in **ValuePtr* is truncated to *BufferLength* minus the length of the null-termination character and is null-terminated by DB2 CLI.

Usage

For a list of attributes that can be set, see "SQLSetConnectAttr - Set Connection Attributes" on page 618. Note that if *Attribute* specifies an attribute that returns a string, *ValuePtr* must be a pointer to a buffer for the string. The maximum length of the string, including the null termination character, will be *BufferLength* bytes.

Depending on the attribute, an application does not need to establish a connection prior to calling `SQLGetConnectAttr()`. However, if `SQLGetConnectAttr()` is called and the specified attribute does not have a default and has not been set by a prior call to `SQLSetConnectAttr()`, `SQLGetConnectAttr()` will return SQL_NO_DATA.

If *Attribute* is SQL_ATTR_ TRACE or SQL_ATTR_ TRACEFILE, *ConnectionHandle* does not have to be valid, and `SQLGetConnectAttr()` will not return SQL_ERROR if *ConnectionHandle* is invalid. These attributes apply to all connections. `SQLGetConnectAttr()` will return SQL_ERROR if another argument is invalid.

While an application can set statement attributes using `SQLSetConnectAttr()`, an application cannot use `SQLGetConnectAttr()` to retrieve statement attribute values; it must call `SQLGetStmtAttr()` to retrieve the setting of statement attributes.

The SQL_ATTR_AUTO_IPD connection attribute can be returned by a call to `SQLGetConnectAttr()`, but cannot be set by a call to `SQLSetConnectAttr()`.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

- SQL_NO_DATA
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 91. SQLGetConnectAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The data returned in **ValuePtr* was truncated to be *BufferLength* minus the length of a null termination character. The the length of the untruncated string value is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	An *Attribute* value was specified that required an open connection.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	`SQLBrowseConnect()` was called for the *ConnectionHandle* and returned SQL_NEED_DATA. This function was called before `SQLBrowseConnect()` returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* was less than 0.
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid connection or statement attribute for the version of the DB2 CLI driver, but was not supported by the data source.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- “SQLGetStmtAttr - Get Current Setting of a Statement Attribute” on page 545
- “SQLSetConnectAttr - Set Connection Attributes” on page 618
- “SQLSetStmtAttr - Set Options Related to a Statement” on page 702

SQLGetConnectOption - Return Current Setting of A Connect Option

Deprecated

Note:

In ODBC version 3, `SQLGetConnectOption()` has been deprecated and replaced with `SQLGetConnectAttr()`; see "SQLGetConnectAttr - Get Current Attribute Setting" on page 439 for more information.

Although this version of DB2 CLI continues to support `SQLGetConnectOption()`, we recommend that you begin using `SQLGetConnectAttr()` in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

SQLGetCursorName - Get Cursor Name

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLGetCursorName() returns the cursor name associated with the input statement handle. If a cursor name was explicitly set by calling SQLSetCursorName(), this name will be returned; otherwise, an implicitly generated name will be returned.

Syntax

```
SQLRETURN   SQLGetCursorName (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLCHAR      *FAR CursorName,        /* szCursor */
                SQLSMALLINT       BufferLength,      /* cbCursorMax */
                SQLSMALLINT  *FAR NameLengthPtr);    /* pcbCursor */
```

Function Arguments

Table 92. SQLGetCursorName Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLCHAR *	*CursorName*	output	Cursor name
SQLSMALLINT	*BufferLength*	input	Length of buffer *CursorName*
SQLSMALLINT *	*NameLengthPtr*	output	Number of bytes available to return for *CursorName*

Usage

SQLGetCursorName() will return the cursor name set explicitly with SQLSetCursorName(), or if no name was set, it will return the cursor name internally generated by DB2 CLI.

If a name is set explicitly using SQLSetCursorName(), this name will be returned until the statement is dropped, or until another explicit name is set.

Internally generated cursor names always begin with SQLCUR or SQL_CUR. Cursor names are always 18 characters or less, and are always unique within a connection.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR

- SQL_INVALID_HANDLE

Diagnostics

Table 93. SQLGetCursorName SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The cursor name returned in *CursorName* was longer than the value in *BufferLength*, and is truncated to *BufferLength* - 1 bytes. The argument *NameLengthPtr* contains the length of the full cursor name available for return. The function returns SQL_SUCCESS_WITH_INFO.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* is less than 0.

Restrictions

ODBC generated cursor names start with SQL_CUR, DB2 CLI generated cursor names start with SQLCUR, and X/Open CLI generated cursor names begin with either SQLCUR or SQL_CUR.

Example

```
/* From CLI sample getcurs.c */
/* ... */
    SQLCHAR * sqlstmt = "SELECT name, job FROM staff "
                        "WHERE job = 'Clerk' "
                        "FOR UPDATE OF job" ;
/* ... */
    rc = SQLExecDirect( hstmt1, sqlstmt, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;
```

```
/* Get Cursor of the SELECT statement's handle */
rc = SQLGetCursorName( hstmt1, cursor, 19, &clength ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;

/* bind name to first column in the result set */
rc = SQLBindCol( hstmt1, 1, SQL_C_CHAR, name.s, 10, &name.ind ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;

/* bind job to second column in the result set */
rc = SQLBindCol( hstmt1, 2, SQL_C_CHAR, job.s, 6, &job.ind ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;

printf( "Job Change for all clerks\n" ) ;

while ((rc = SQLFetch(hstmt1)) == SQL_SUCCESS) {
    printf("Name: %-9.9s Job: %-5.5s \n", name.s, job.s);
    printf("Enter new job or return to continue\n");
    gets((char *)newjob);
    if (newjob[0] != '\0') {
        sprintf((char *)updstmt,
                "UPDATE staff set job = '%s' where current of %s",
                newjob, cursor);
        rc2 = SQLExecDirect(hstmt2, updstmt, SQL_NTS);
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt2, rc2 ) ;
    }
}

if ( rc != SQL_NO_DATA_FOUND )
   CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;
```

References

- “SQLExecute - Execute a Statement” on page 373
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLPrepare - Prepare a Statement” on page 583
- “SQLSetCursorName - Set Cursor Name” on page 645

SQLGetData - Get Data From a Column

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLGetData() retrieves data for a single column in the current row of the result set. This is an alternative to SQLBindCol(), which is used to transfer data directly into application variables or LOB locators on each SQLFetch() or SQLFetchScroll() call. SQLGetData() can also be used to retrieve large data values in pieces.

SQLFetch() must be called before SQLGetData().

After calling SQLGetData() for each column, SQLFetch() or SQLFetchScroll() is called to retrieve the next row.

Syntax

```
SQLRETURN   SQLGetData       (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLUSMALLINT      ColumnNumber,      /* icol */
                SQLSMALLINT       TargetType,        /* fCType */
                SQLPOINTER        TargetValuePtr,    /* rgbValue */
                SQLINTEGER        BufferLength,      /* cbValueMax */
                SQLINTEGER   *FAR StrLen_or_IndPtr); /* pcbValue */
```

Function Arguments

Table 94. SQLGetData Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLUSMALLINT	*ColumnNumber*	input	Column number for which the data retrieval is requested. Result set columns are numbered sequentially. • Column numbers start at 1 if bookmarks are not used (SQL_ATTR_USE_BOOKMARKS statement attribute set to SQL_UB_OFF). • Column numbers start at 0 if bookmarks are used (the statement attribute set to SQL_UB_ON or SQL_UB_VARIABLE).

Table 94. SQLGetData Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	*TargetType*	input	The C data type of the column identifier by *ColumnNumber*. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_NUMERIC [a] • SQL_C_SBIGINT • SQL_C_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_UBIGINT Specifying SQL_C_DEFAULT results in the data being converted to its default C data type, refer to Table 2 on page 29 for more information.
SQLPOINTER	*TargetValuePtr*	output	Pointer to buffer where the retrieved column data is to be stored.
SQLINTEGER	*BufferLength*	input	Maximum size of the buffer pointed to by *TargetValuePtr*

Table 94. SQLGetData Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER *	*StrLen_or_IndPtr*	output	Pointer to value which indicates the number of bytes DB2 CLI has available to return in the *TargetValuePtr* buffer. If the data is being retrieved in pieces, this contains the number of bytes still remaining. The value is SQL_NULL_DATA if the data value of the column is null. If this pointer is NULL and SQLFetch() has obtained a column containing null data, then this function will fail because it has no means of reporting this. If SQLFetch() has fetched a column containing binary data, then the pointer to *StrLen_or_IndPtr* must not be NULL or this function will fail because it has no other means of informing the application about the length of the data retrieved in the *TargetValuePtr* buffer.

Note: DB2 CLI will provide some performance enhancement if *TargetValuePtr* is placed consecutively in memory after *StrLen_or_IndPtr*

Usage

SQLGetData() can be used with SQLBindCol() for the same result set, as long as SQLFetch() and not SQLFetchScroll() is used. The general steps are:

1. SQLFetch() - advances cursor to first row, retrieves first row, transfers data for bound columns.
2. SQLGetData() - transfers data for the specified column.
3. Repeat step 2 for each column needed.
4. SQLFetch() - advances cursor to next row, retrieves next row, transfers data for bound columns.
5. Repeat steps 2, 3 and 4 for each row in the result set, or until the result set is no longer needed.

SQLGetData() can also be used to retrieve long columns if the C data type (*TargetType*) is SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR, or if *TargetType* is SQL_C_DEFAULT and the column type denotes a binary or character string.

Upon each SQLGetData() call, if the data available for return is greater than or equal to *BufferLength*, truncation occurs. Truncation is indicated by a function return code of SQL_SUCCESS_WITH_INFO coupled with a SQLSTATE denoting data truncation. The application can call SQLGetData() again, with

the same ColumnNumber value, to get subsequent data from the same unbound column starting at the point of truncation. To obtain the entire column, the application repeats such calls until the function returns SQL_SUCCESS. The next call to SQLGetData() returns SQL_NO_DATA_FOUND.

Although SQLGetData() can be used for the sequential retrieval of LOB column data, use the DB2 CLI LOB functions if only a portion of the LOB data or a few sections of the LOB column data are needed:

1. Bind the column to a LOB locator.
2. Fetch the row.
3. Use the locator in a SQLGetSubString() call, to retrieve the data in pieces (SQLGetLength() and SQLGetPosition() may also be required in order to determine the values of some of the arguments).
4. Repeat step 2.

Truncation is also affected by the SQL_ATTR_MAX_LENGTH statement attribute. The application can specify that truncation is not to be reported by calling SQLSetStmtAttr() with SQL_ATTR_MAX_LENGTH and a value for the maximum length to return for any one column, and by allocating a *TargetValuePtr* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS will be returned and the maximum length, not the actual length will be returned in *StrLen_or_IndPtr*.

To discard the column data part way through the retrieval, the application can call SQLGetData() with *ColumnNumber* set to the next column position of interest. To discard data that has not been retrieved for the entire row, the application should call SQLFetch() to advance the cursor to the next row; or, if it is not interested in any more data from the result set, call SQLFreeStmt() to close the cursor.

The *TargetType* input argument determines the type of data conversion (if any) needed before the column data is placed into the storage area pointed to by *TargetValuePtr*.

For SQL graphic column data:

- The length of the *TargetValuePtr* buffer (*BufferLength*) should be a multiple of 2. The application can determine the SQL data type of the column by first calling SQLDescribeCol() or SQLColAttribute().
- The pointer to *StrLen_or_IndPtr* must not be NULL since DB2 CLI will be storing the number of octets stored in *TargetValuePtr*.
- If the data is to be retrieved in piecewise fashion, DB2 CLI will attempt to fill *TargetValuePtr* to the nearest multiple of two octets that is still less than

or equal to *BufferLength*. This means if *BufferLength* is not a multiple of two, the last byte in that buffer will be untouched; DB2 CLI will not split a double-byte character.

The contents returned in *TargetValuePtr* is always null-terminated unless the column data to be retrieved is binary, or if the SQL data type of the column is graphic (DBCS) and the C buffer type is SQL_C_CHAR. If the application is retrieving the data in multiple chunks, it should make the proper adjustments (for example, strip off the null-terminator before concatenating the pieces back together assuming the null termination environment attribute is in effect).

Truncation of numeric data types is reported as a warning if the truncation involves digits to the right of the decimal point. If truncation occurs to the left of the decimal point, an error is returned (refer to the diagnostics section).

With the exception of scrollable cursors, applications that use SQLFetchScroll() to retrieve data should call SQLGetData() only when the rowset size is 1 (equivalent to issuing SQLFetch()). SQLGetData() can only retrieve column data for a row where the cursor is currently positioned.

Using SQLGetData() with Scrollable Cursors

SQLGetData() can also be used with scrollable cursors. You can save a pointer to any row in the result set; a bookmark. The application can then use that bookmark as a relative position to retrieve a rowset of information.

Once you have positioned the cursor to a row in a rowset using SQLSetPos(), you can obtain the bookmark value from column 0 using SQLGetData(). In most cases you will not want to bind column 0 and retrieve the bookmark value for every row, but use SQLGetData() to retrieve the bookmark value for the specific row you require.

See “Scrollable Cursors” on page 68 for more information.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND is returned when the preceding SQLGetData() call has retrieved all of the data for this column.

SQL_SUCCESS is returned if a zero-length string is retrieved by SQLGetData(). If this is the case, StrLen_or_IndPtr will contain 0, and TargetValuePtr will contain a null terminator.

If the preceding call to SQLFetch() failed, SQLGetData() should not be called since the result is undefined.

Diagnostics

Table 95. SQLGetData SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	Data returned for the specified column (ColumnNumber) was truncated. String or numeric values are right truncated. SQL_SUCCESS_WITH_INFO is returned.
07006	Invalid conversion.	The data value cannot be converted to the C data type specified by the argument *TargetType*. The function has been called before for the same *ColumnNumber* value but with a different *TargetType* value.
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument StrLen_or_IndPtr was a null pointer and the value of the column is null. There is no means to report SQL_NULL_DATA.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for the column would have caused the whole part of the number to be truncated.
22005	Error in assignment.	A returned value was incompatible with the data type denoted by the argument *TargetType*.
22007	Invalid datetime format.	Conversion from character a string to a datetime format was indicated, but an invalid string representation or value was specified, or the value was an invalid date.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
24000	Invalid cursor state.	The previous SQLFetch() resulted in SQL_ERROR or SQL_NO_DATA found; as a result, the cursor is not positioned on a row.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.

Table 95. SQLGetData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY002	Invalid column number.	The specified column was less than 0 or greater than the number of result columns. The specified column was 0, but the application did not enable bookmarks (by setting the SQL_ATTR_USE_BOOKMARKS statement attribute). `SQLExtendedFetch()` was called for this result set.
HY003	Program type out of range.	*TargetType* was not a valid data type or SQL_C_DEFAULT.
HY010	Function sequence error.	The specified *StatementHandle* was not in a cursor positioned state. The function was called without first calling `SQLFetch()`. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value of the argument *BufferLength* is less than 0 and the argument *TargetType* is SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR or (SQL_C_DEFAULT and the default type is one of SQL_C_CHAR, SQL_C_BINARY, or SQL_C_DBCHAR).
HYC00	Driver not capable.	The SQL data type for the specified data type is recognized but not supported by DB2 CLI. The requested conversion from the SQL data type to the application data *TargetType* cannot be performed by DB2 CLI or the data source. The column was bound using `SQLBindFileToCol()`.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Restrictions

None.

Example

Refer to “Example” on page 406 for a comparison between using bound columns and using SQLGetData().

```
/* From CLI sample getdata.c */
/* ... */
    SQLCHAR * sqlstmt = "SELECT deptname, location from org "
                        "WHERE division = 'Eastern'" ;
/* ... */

    rc = SQLExecDirect( hstmt, sqlstmt, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf( "Departments in Eastern division:\n" ) ;
    printf( "DEPTNAME       Location\n" ) ;
    printf( "-------------- ------------\n" ) ;

    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        rc = SQLGetData(hstmt, 1, SQL_C_CHAR, deptname.s,
                        15, &(deptname.ind));
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
        rc = SQLGetData(hstmt, 2, SQL_C_CHAR, location.s,
                        15, &(location.ind));
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
        printf("%-14.14s %-14.14s \n", deptname.s, location.s);
    }

    if ( rc != SQL_NO_DATA_FOUND )
       CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

References

- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408
- “SQLFetch - Fetch Next Row” on page 396
- “SQLGetSubString - Retrieve Portion of A String Value” on page 550

SQLGetDataLinkAttr - Get DataLink Attribute Value

Purpose

Specification:	DB2 CLI 5.2		ISO CLI

Return the current value of an attribute of a datalink value.

Syntax

```
SQLRETURN   SQLGetDataLinkAttr(SQLHSTMT          StatementHandle,
                               SQLSMALLINT       Attribute,
                               SQLCHAR     FAR   *DataLink,
                               SQLINTEGER        DataLinkLength,
                               SQLPOINTER        *ValuePtr,
                               SQLINTEGER        BufferLength,
                               SQLINTEGER  FAR   *StringLengthPtr);
```

Function Arguments

Table 96. SQLGetDataLinkAttr Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Used only for diagnostic reporting.
SQLSMALLINT	Attribute	input	Identifies the attribute of the DataLink that is to be extracted. Possible values are: • SQL_ATTR_DATALINK_COMMENT • SQL_ATTR_DATALINK_LINKTYPE • SQL_ATTR_DATALINK_URLCOMPLETE (complete URL to access a file) • SQL_ATTR_DATALINK_URLPATH (to access a file within a file server) • SQL_ATTR_DATALINK_URLPATHONLY (file path only) • SQL_ATTR_DATALINK_URLSCHEME • SQL_ATTR_DATALINK_URLSERVER
SQLCHAR *	DataLink	input	The DATALINK value from which the attribute is to be extracted.
SQLINTEGER	DataLinkLength	input	The length of the DATALINK value.
SQLPOINTER *	ValuePtr	output	A pointer to memory in which to return the value of the attribute specified by *Attribute*.
SQLINTEGER	BufferLength	input	Length of the Attribute buffer.

Table 96. SQLGetDataLinkAttr Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER	*StringLength	output	A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character) available to return in **Attribute*. If *Attribute* is a null pointer, no length is returned. If the number of bytes available to return is greater than BufferLength minus the length of the null-termination character, then SQLSTATE HY090 is returned.

Usage

The function is used with a DATALINK value that was retrieved from the database or built using `SQLBuildDataLink()`. The *AttrType* value determines the attribute from the DATALINK value that is returned. The maximum length of the string, including the null termination character, will be *BufferLength* bytes.

Refer to the *Administration Guide, Design and Implementation* for more information on Data Links.

Return Codes

- SQL_SUCCESS
- SQL_NO_DATA
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 97. SQLExtendedBind() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
01004	Data truncated.	The data returned in **ValuePtr* was truncated to be *BufferLength* minus the length of the null termination character. The length of the untruncated string value is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)

Table 97. SQLExtendedBind() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY009	Invalid argument value.	The value specified for the argument **DataLink* was a null pointer or was not valid.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* was less than 0 or the values specified for the argument *DataLinkLength* was less than 0 and not equal to SQL_NTS.
HY092	Option type out of range.	The value specified for the argument *AttrType* was not valid.

Restrictions

None.

Example

See “Example” on page 273.

References

- “SQLBuildDataLink - Build DATALINK Value” on page 272

SQLGetDescField - Get Single Field Settings of Descriptor Record

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLGetDescField() returns the current settings of a single field of a descriptor record.

Syntax

```
SQLRETURN   SQLGetDescField  (SQLHDESC          DescriptorHandle,
                              SQLSMALLINT       RecNumber,
                              SQLSMALLINT       FieldIdentifier,
                              SQLPOINTER        ValuePtr,
                              SQLINTEGER        BufferLength,
                              SQLINTEGER        *StringLengthPtr);
```

Function Arguments

Table 98. SQLGetDescField Arguments

Data Type	Argument	Use	Description
SQLHDESC	*DescriptorHandle*	input	Descriptor handle.
SQLSMALLINT	*RecNumber*	input	Indicates the descriptor record from which the application seeks information. Descriptor records are numbered from 0, with record number 0 being the bookmark record. If the *FieldIdentifier* argument indicates a field of the descriptor header record, *RecNumber* must be 0. If *RecNumber* is less than SQL_DESC_COUNT, but the row does not contain data for a column or parameter, a call to SQLGetDescField() will return the default values of the fields. For more information, see SQLSetDescField() "Initialization of Descriptor Fields" on page 651.
SQLSMALLINT	*FieldIdentifier*	input	Indicates the field of the descriptor whose value is to be returned. For more information, see SQLSetDescField() "FieldIdentifier Arguments" on page 659.
SQLPOINTER	ValuePtr	output	Pointer to a buffer in which to return the descriptor information. The data type depends on the value of *FieldIdentifier*.

Table 98. SQLGetDescField Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER	BufferLength	input	• If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*. • If *ValuePtr* is a pointer, but not to a string, then *BufferLength* should have the value SQL_IS_POINTER. • If *ValuePtr* is not a pointer, then *BufferLength* should have the value SQL_IS_NOT_POINTER. • If the value in **ValuePtr* is of a unicode data type the *BufferLength* argument must be an even number.
SQLSMALLINT	*StringLengthPtr	output	Pointer to the total number of bytes (excluding the number of bytes required for the null termination character) available to return in **ValuePtr*.

Usage

An application can call SQLGetDescField() to return the value of a single field of a descriptor record. A call to SQLGetDescField() can return the setting of any field in any descriptor type, including header fields, record fields, and bookmark fields. An application can obtain the settings of multiple fields in the same or different descriptors, in arbitrary order, by making repeated calls to SQLGetDescField(). SQLGetDescField() can also be called to return DB2 CLI defined descriptor fields.

For performance reasons, an application should not call SQLGetDescField() for an IRD before executing a statement.

The settings of multiple fields that describe the name, data type, and storage of column or parameter data can also be retrieved in a single call to SQLGetDescRec(). SQLGetStmtAttr() can be called to return the setting of a single field in the descriptor header that is also a statement attribute.

When an application calls SQLGetDescField() to retrieve the value of a field that is undefined for a particular descriptor type, the function returns SQLSTATE HY091 (Invalid descriptor field identifier). When an application calls SQLGetDescField() to retrieve the value of a field that is defined for a particular descriptor type, but has no default value and has not been set yet, the function returns SQL_SUCCESS but the value returned for the field is undefined. For more information, see SQLSetDescField() “Initialization of Descriptor Fields” on page 651.

The SQL_DESC_ALLOC_TYPE header field is available as read-only. This field is defined for all types of descriptors.

The following record fields are available as read-only. Each of these fields is defined either for the IRD only, or for both the IRD and the IPD.

```
SQL_DESC_AUTO_UNIQUE_VALUE    SQL_DESC_LITERAL_SUFFIX
SQL_DESC_BASE_COLUMN_NAME     SQL_DESC_LOCAL_TYPE_NAME
SQL_DESC_CASE_SENSITIVE       SQL_DESC_SCHEMA_NAME
SQL_DESC_CATALOG_NAME         SQL_DESC_SEARCHABLE
SQL_DESC_DISPLAY_SIZE         SQL_DESC_TABLE_NAME
SQL_DESC_FIXED_PREC_SCALE     SQL_DESC_TYPE_NAME
SQL_DESC_LABEL                SQL_DESC_UNSIGNED
SQL_DESC_LITERAL_PREFIX       SQL_DESC_UPDATABLE
```

For a description of the above fields, and fields that can be set in a descriptor header or record, see the SQLSetDescField() section. For more information on descriptors, see “Using Descriptors” on page 97.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_NO_DATA
- SQL_INVALID_HANDLE

SQL_NO_DATA is returned if *RecNumber* is greater than the number of descriptor records.

SQL_NO_DATA is returned if *DescriptorHandle* is an IRD handle and the statement is in the prepared or executed state, but there was no open cursor associated with it.

Diagnostics

Table 99. SQLGetDescField SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
HY013	Unexpected memory handling error.	The *HandleType* argument was SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC; and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY021	Inconsistent descriptor information.	The descriptor information checked during a consistency check was not consistent. For more information, see SQLSetDescField() “Consistency Checks” on page 674.

Table 99. SQLGetDescField SQLSTATEs (continued)

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer **ValuePtr* was not large enough to return the entire descriptor field, so the field was truncated. The length of the untruncated descriptor field is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
07009	Invalid descriptor index.	The value specified for the *RecNumber* argument was less than 1, the SQL_ATTR_USE_BOOKMARK statement attribute was SQL_UB_OFF, and the field was not a header field or a DB2 CLI defined field. The *FieldIdentifier* argument was a record field, and the *RecNumber* argument was 0. The *RecNumber* argument was less than 0, and the field was not a header field or a DB2 CLI defined field.
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was connected failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate the memory required to support execution or completion of the function.
HY007	Associated statement is not prepared.	*DescriptorHandle* was associated with an IRD, and the associated statement handle was not in the prepared or executed state.
HY010	Function sequence error.	*DescriptorHandle* was associated with a StatementHandle for which an asynchronously executing function (not this one) was called and was still executing when this function was called. *DescriptorHandle* was associated with a *StatementHandle* for which `SQLExecute()` or `SQLExecDirect()` was called and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.

Table 99. SQLGetDescField SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY091	Descriptor type out of range.	*FieldIdentifier* was undefined for the *DescriptorHandle*. The value specified for the *RecNumber* argument was greater than the value in the SQL_DESC_COUNT field.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLGetDescRec - Get Multiple Field Settings of Descriptor Record" on page 463
- "SQLSetDescField - Set a Single Field of a Descriptor Record" on page 649
- "SQLSetDescRec - Set Multiple Descriptor Fields for a Column or Parameter Data" on page 677

SQLGetDescRec - Get Multiple Field Settings of Descriptor Record

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLGetDescRec() returns the current settings of multiple fields of a descriptor record. The fields returned describe the name, data type, and storage of column or parameter data.

Syntax

```
SQLRETURN   SQLGetDescRec    (SQLHDESC          DescriptorHandle,
                              SQLSMALLINT       RecNumber,
                              SQLCHAR           *Name,
                              SQLSMALLINT       BufferLength,
                              SQLSMALLINT       *StringLengthPtr,
                              SQLSMALLINT       *TypePtr,
                              SQLSMALLINT       *SubTypePtr,
                              SQLINTEGER        *LengthPtr,
                              SQLSMALLINT       *PrecisionPtr,
                              SQLSMALLINT       *ScalePtr,
                              SQLSMALLINT       *NullablePtr);
```

Function Arguments

Table 100. SQLGetDescRec Arguments

Data Type	Argument	Use	Description
SQLHDESC	DescriptorHandle	input	Descriptor handle.
SQLSMALLINT	*RecNumber*	input	Indicates the descriptor record from which the application seeks information. Descriptor records are numbered from 0, with record number 0 being the bookmark record. The *RecNumber* argument must be less than or equal to the value of SQL_DESC_COUNT. If *RecNumber* is less than SQL_DESC_COUNT, but the row does not contain data for a column or parameter, a call to SQLGetDescRec() will return the default values of the fields (for more information, see "Initialization of Descriptor Fields" in SQLSetDescField()).
SQLCHAR	*Name*	output	A pointer to a buffer in which to return the SQL_DESC_NAME field for the descriptor record.
SQLINTEGER	BufferLength	input	Length of the **Name* buffer, in bytes.

Table 100. SQLGetDescRec Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	*StringLengthPtr	output	A pointer to a buffer in which to return the number of bytes of data available to return in the **Name* buffer, excluding the null termination character. If the number of bytes was greater than or equal to *BufferLength*, the data in **Name* is truncated to *BufferLength* minus the length of a null termination character, and is null terminated by DB2 CLI.
SQLSMALLINT	*TypePtr*	output	A pointer to a buffer in which to return the value of the SQL_DESC_TYPE field for the descriptor record.
SQLSMALLINT	*SubTypePtr*	output	For records whose type is SQL_DATETIME or SQL_INTERVAL, this is a pointer to a buffer in which to return the value of the SQL_DESC_DATETIME_INTERVAL_CODE field.
SQLINTEGER	*LengthPtr*	output	A pointer to a buffer in which to return the value of the SQL_DESC_OCTET_LENGTH field for the descriptor record.
SQLSMALLINT	*PrecisionPtr*	output	A pointer to a buffer in which to return the value of the SQL_DESC_PRECISION field for the descriptor record.
SQLSMALLINT	*ScalePtr*	output	A pointer to a buffer in which to return the value of the SQL_DESC_SCALE field for the descriptor record.
SQLSMALLINT	*NullablePtr	output	A pointer to a buffer in which to return the value of the SQL_DESC_NULLABLE field for the descriptor record.

Usage

An application can call `SQLGetDescRec()` to retrieve the values of the following fields for a single column or parameter:

- SQL_DESC_NAME
- SQL_DESC_TYPE
- SQL_DESC_DATETIME_INTERVAL_CODE (for records whose type is SQL_DATETIME)
- SQL_DESC_OCTET_LENGTH
- SQL_DESC_PRECISION
- SQL_DESC_SCALE
- SQL_DESC_NULLABLE

SQLGetDescRec() does not retrieve the values for header fields.

An application can inhibit the return of a field's setting by setting the argument corresponding to the field to a null pointer. When an application calls SQLGetDescRec() to retrieve the value of a field that is undefined for a particular descriptor type, the function returns SQL_SUCCESS but the value returned for the field is undefined. For example, calling SQLGetDescRec() for the SQL_DESC_NAME or SQL_DESC_NULLABLE field of an APD or ARD will return SQL_SUCCESS but an undefined value for the field.

When an application calls SQLGetDescRec() to retrieve the value of a field that is defined for a particular descriptor type, but has no default value and has not been set yet, the function returns SQL_SUCCESS but the value returned for the field is undefined.

The values of fields can also be retrieved individually by a call to SQLGetDescField(). For a description of the fields in a descriptor header or record, see "SQLSetDescField - Set a Single Field of a Descriptor Record" on page 649. For more information on descriptors, see "Using Descriptors" on page 97.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_NO_DATA
- SQL_INVALID_HANDLE

SQL_NO_DATA is returned if *RecNumber* is greater than the number of descriptor records.

SQL_NO_DATA is returned if *DescriptorHandle* is an IRD handle and the statement in in the prepared or executed state, but there was no open cursor associated with it.

Diagnostics

Table 101. SQLGetDescRec SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)

Table 101. SQLGetDescRec SQLSTATEs (continued)

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer **Name* was not large enough to return the entire descriptor field, so the field was truncated. The length of the untruncated descriptor field is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
07009	Invalid descriptor index.	The *RecNumber* argument was set to 0 and the *DescriptorHandle* argument was an IPD handle. The *RecNumber* argument was set to 0, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF. The *RecNumber* argument was less than 0.
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was connected failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate the memory required to support execution or completion of the function.
HY007	Associated statement is not prepared.	*DescriptorHandle* was associated with an IRD, and the associated statement handle was not in the prepared or executed state.
HY010	Function sequence error.	*DescriptorHandle* was associated with a *StatementHandle* for which an asynchronously executing function (not this one) was called and was still executing when this function was called. *DescriptorHandle* was associated with a *StatementHandle* for which `SQLExecute()` or `SQLExecDirect()` was called and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLSetDescRec - Set Multiple Descriptor Fields for a Column or Parameter Data" on page 677
- "SQLGetDescField - Get Single Field Settings of Descriptor Record" on page 458
- "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227
- "SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator" on page 247

SQLGetDiagField - Get a Field of Diagnostic Data

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLGetDiagField() returns the current value of a field of a diagnostic data structure, associated with a specific handle, that contains error, warning, and status information.

Syntax

```
SQLRETURN   SQLGetDiagField  (SQLSMALLINT       HandleType,
                              SQLHANDLE         Handle,
                              SQLSMALLINT       RecNumber,
                              SQLSMALLINT       DiagIdentifier,
                              SQLPOINTER        DiagInfoPtr,
                              SQLSMALLINT       BufferLength,
                              SQLSMALLINT       *StringLengthPtr);
```

Function Arguments

Table 102. SQLGetDiagField Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	A handle type identifier that describes the type of handle for which diagnostics are desired. Must be one of the following: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC
SQLHANDLE	*Handle*	input	A handle for the diagnostic data structure, of the type indicated by *HandleType*.
SQLSMALLINT	*RecNumber*	input	Indicates the status record from which the application seeks information. Status records are numbered from 1. If the *DiagIdentifier* argument indicates any field of the diagnostics header record, *RecNumber* must be 0. If not, it should be greater than 0.
SQLSMALLINT	*DiagIdentifier*	input	Indicates the field of the diagnostic data structure whose value is to be returned. For more information, see “DiagIdentifier Argument” on page 470.
SQLPOINTER	*DiagInfoPtr*	output	Pointer to a buffer in which to return the diagnostic information. The data type depends on the value of *DiagIdentifier*.

Table 102. SQLGetDiagField Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER	BufferLength	input	If *DiagInfoPtr* points to a character string, this argument should be the length of *ValuePtr. If ValuePtr is a pointer, but not to a string, then *BufferLength* should have the value SQL_IS_POINTER. If ValuePtr is not a pointer, then *BufferLength* should have the value SQL_IS_NOT_POINTER. If the value in **DiagInfoPtr* is a unicode string the *BufferLength* argument must be an even number.
SQLSMALLINT	*StringLengthPtr	output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes required for the null termination character) available to return in **DiagInfoPtr*, for character data. If the number of bytes available to return is greater than *BufferLength*, then the text in **DiagInfoPtr* is truncated to *BufferLength* minus the length of a null termination character. This argument is ignored for non-character data.

Usage

An application typically calls SQLGetDiagField() to accomplish one of three goals:

1. To obtain specific error or warning information when a function call has returned SQL_ERROR or SQL_SUCCESS_WITH_INFO (or SQL_NEED_DATA for the SQLBrowseConnect() function).
2. To find out the number of rows in the data source that were affected when insert, delete, or update operations were performed with a call to SQLExecute() or SQLExecDirect() (from the SQL_DIAG_ROW_COUNT header field), or to find out the number of rows that exist in the current open static scrollable cursor (from the SQL_DIAG_CURSOR_ROW_COUNT header field).
3. To determine which function was executed by a call to SQLExecDirect() or SQLExecute() (from the SQL_DIAG_DYNAMIC_FUNCTION and SQL_DIAG_DYNAMIC_FUNCTION_CODE header fields).

Any DB2 CLI function can post zero or more errors each time it is called, so an application can call SQLGetDiagField() after any function call. SQLGetDiagField() retrieves only the diagnostic information most recently associated with the diagnostic data structure specified in the *Handle* argument. If the application calls another function, any diagnostic information from a previous call with the same handle is lost.

An application can scan all diagnostic records by incrementing *RecNumber*, as long as SQLGetDiagField() returns SQL_SUCCESS. The number of status records is indicated in the SQL_DIAG_NUMBER header field. Calls to SQLGetDiagField() are non-destructive as far as the header and status records are concerned. The application can call SQLGetDiagField() again at a later time to retrieve a field from a record, as long as another function other than SQLGetDiagField(), SQLGetDiagRec(), or SQLError() has not been called in the interim, which would post records on the same handle.

An application can call SQLGetDiagField() to return any diagnostic field at any time, with the exception of SQL_DIAG_ROW_COUNT, which will return SQL_ERROR if *Handle* was not a statement handle on which an SQL statement had been executed. If any other diagnostic field is undefined, the call to SQLGetDiagField() will return SQL_SUCCESS (provided no other error is encountered), and an undefined value is returned for the field.

HandleType Argument

Each handle type can have diagnostic information associated with it. The *HandleType* argument denotes the handle type of *Handle*.

Some header and record fields cannot be returned for all types of handles: environment, connection, statement, and descriptor. Those handles for which a field is not applicable are indicated in the Header Field and Record Fields sections below.

No DB2 CLI specific header diagnostic field should be associated with an environment handle.

DiagIdentifier Argument

This argument indicates the identifier of the field desired from the diagnostic data structure. If *RecNumber* is greater than or equal to 1, the data in the field describes the diagnostic information returned by a function. If *RecNumber* is 0, the field is in the header of the diagnostic data structure, so contains data pertaining to the function call that returned the diagnostic information, not the specific information.

Header Fields

The following header fields can be included in the *DiagIdentifier* argument. The only diagnostic header fields that are defined for a descriptor field are SQL_DIAG_NUMBER and SQL_DIAG_RETURNCODE.

Table 103. Header Fields for DiagIdentifier Arguments

SQL_DIAG_CURSOR_ROW_COUNT (return type SQLINTEGER)

Table 103. Header Fields for DiagIdentifier Arguments (continued)

This field contains the count of rows in the cursor. Its semantics depend upon the SQLGetInfo() information types:

- SQL_DYNAMIC_CURSOR_ATTRIBUTES2
- SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2
- SQL_KEYSET_CURSOR_ATTRIBUTES2
- SQL_STATIC_CURSOR_ATTRIBUTES2

which indicate which row counts are available for each cursor type (in the SQL_CA2_CRC_EXACT and SQL_CA2_CRC_APPROXIMATE bits).

The contents of this field are defined only for statement handles and only after SQLExecute(), SQLExecDirect(), or SQLMoreResults() has been called. Calling SQLGetDiagField() with a *DiagIdentifier* of SQL_DIAG_CURSOR_ROW_COUNT on other than a statement handle will return SQL_ERROR.

SQL_DIAG_DYNAMIC_FUNCTION (return type CHAR *)

This is a string that describes the SQL statement that the underlying function executed (see "Dynamic Function Fields" on page 473 for the values that DB2 CLI supports). The contents of this field are defined only for statement handles, and only after a call to SQLExecute() or SQLExecDirect(). The value of this field is undefined before a call to SQLExecute() or SQLExecDirect().

SQL_DIAG_DYNAMIC_FUNCTION_CODE (return type SQLINTEGER)

This is a numeric code that describes the SQL statement that was executed by the underlying function (see "Dynamic Function Fields" on page 473 for the values that DB2 CLI supports). The contents of this field are defined only for statement handles, and only after a call to SQLExecute() or SQLExecDirect(). The value of this field is undefined before a call to SQLExecute(), SQLExecDirect(), or SQLMoreResults(). Calling SQLGetDiagField() with a *DiagIdentifier* of SQL_DIAG_DYNAMIC_FUNCTION_CODE on other than a statement handle will return SQL_ERROR. The value of this field is undefined before a call to SQLExecute() or SQLExecDirect().

SQL_DIAG_NUMBER (return type SQLINTEGER)

The number of status records that are available for the specified handle.

SQL_DIAG_RETURNCODE (return type RETCODE)

Table 103. Header Fields for DiagIdentifier Arguments (continued)

Return code returned by the last executed function associated with the specified handle. See "Function Return Codes" on page 26 for a list of return codes. If no function has yet been called on the *Handle*, SQL_SUCCESS will be returned for SQL_DIAG_RETURNCODE.

SQL_DIAG_ROW_COUNT (return type SQLINTEGER)

The number of rows affected by an insert, delete, or update performed by SQLExecute(), SQLExecDirect(), or SQLSetPos(). It is defined after a cursor specification has been executed. The contents of this field are defined only for statement handles. The data in this field is returned in the RowCountPtr argument of SQLRowCount(). The data in this field is reset after every function call, whereas the row count returned by SQLRowCount() remains the same until the statement is set back to the prepared or allocated state.

Record Fields

The following record fields can be included in the *DiagIdentifier* argument:

Table 104. Record Fields for DiagIdentifier Arguments

SQL_DIAG_CLASS_ORIGIN (return type CHAR *)

A string that indicates the document that defines the class and subclass portion of the SQLSTATE value in this record.

DB2 CLI always returns an empty string for SQL_DIAG_CLASS_ORIGIN.

SQL_DIAG_COLUMN_NUMBER (return type SQLINTEGER)

If the SQL_DIAG_ROW_NUMBER field is a valid row number in a rowset or set of parameters, then this field contains the value that represents the column number in the result set. Result set column numbers always start at 1; if this status record pertains to a bookmark column, then the field can be zero. It has the value SQL_NO_COLUMN_NUMBER if the status record is not associated with a column number. If DB2 CLI cannot determine the column number that this record is associated with, this field has the value SQL_COLUMN_NUMBER_UNKNOWN. The contents of this field are defined only for statement handles.

SQL_DIAG_CONNECTION_NAME (return type CHAR *)

A string that indicates the name of the connection that the diagnostic record relates to.

DB2 CLI always returns an empty string for SQL_DIAG_CONNECTION_NAME

SQL_DIAG_MESSAGE_TEXT (return type CHAR *)

Table 104. Record Fields for DiagIdentifier Arguments (continued)

An informational message on the error or warning.

SQL_DIAG_NATIVE (return type SQLINTEGER)

A driver / data-source-specific native error code. If there is no native error code, the driver returns 0.

SQL_DIAG_ROW_NUMBER (return type SQLINTEGER)

This field contains the row number in the rowset, or the parameter number in the set of parameters, with which the status record is associated. This field has the value SQL_NO_ROW_NUMBER if this status record is not associated with a row number. If DB2 CLI cannot determine the row number that this record is associated with, this field has the value SQL_ROW_NUMBER_UNKNOWN. The contents of this field are defined only for statement handles.

SQL_DIAG_SERVER_NAME (return type CHAR *)

A string that indicates the server name that the diagnostic record relates to. It is the same as the value returned for a call to `SQLGetInfo()` with the SQL_DATA_SOURCE_NAME InfoType. For diagnostic data structures associated with the environment handle and for diagnostics that do not relate to any server, this field is a zero-length string.

SQL_DIAG_SQLSTATE (return type CHAR *)

A five-character SQLSTATE diagnostic code.

SQL_DIAG_SUBCLASS_ORIGIN (return type CHAR *)

A string with the same format and valid values as SQL_DIAG_CLASS_ORIGIN, that identifies the defining portion of the subclass portion of the SQLSTATE code.

DB2 CLI always returns an empty string for SQL_DIAG_SUBCLASS_ORIGIN.

Values of the Dynamic Function Fields

The table below describes the values of SQL_DIAG_DYNAMIC_FUNCTION and SQL_DIAG_DYNAMIC_FUNCTION_CODE that apply to each type of SQL statement executed by a call to `SQLExecute()` or `SQLExecDirect()`. This is the list that DB2 CLI uses. ODBC also specifies other values.

Table 105. Values of Dynamic Function Fields

SQL Statement Executed	Value of SQL_DIAG_ DYNAMIC_FUNCTION	Value of SQL_DIAG_DYNAMIC_ FUNCTION_CODE
alter-table-statement	“ALTER TABLE”	SQL_DIAG_ALTER_TABLE
create-index-statement	“CREATE INDEX”	SQL_DIAG_CREATE_INDEX
create-table-statement	“CREATE TABLE”	SQL_DIAG_CREATE_TABLE
create-view-statement	“CREATE VIEW”	SQL_DIAG_CREATE_VIEW
cursor-specification	“SELECT CURSOR”	SQL_DIAG_SELECT_CURSOR
delete-statement-positioned	“DYNAMIC DELETE CURSOR”	SQL_DIAG_DYNAMIC_DELETE_ CURSOR
delete-statement-searched	“DELETE WHERE”	SQL_DIAG_DELETE_WHERE
drop-index-statement	“DROP INDEX”	SQL_DIAG_DROP_INDEX
drop-table-statement	“DROP TABLE”	SQL_DIAG_DROP_TABLE
drop-view-statement	“DROP VIEW”	SQL_DIAG_DROP_VIEW
grant-statement	“GRANT”	SQL_DIAG_GRANT
insert-statement	“INSERT”	SQL_DIAG_INSERT
ODBC-procedure-extension	“CALL”	SQL_DIAG_PROCEDURE_CALL
revoke-statement	“REVOKE”	SQL_DIAG_REVOKE
update-statement-positioned	“DYNAMIC UPDATE CURSOR”	SQL_DIAG_DYNAMIC_UPDATE_ CURSOR
update-statement-searched	“UPDATE WHERE”	SQL_DIAG_UPDATE_WHERE
Unknown	empty string	SQL_DIAG_UNKNOWN_STATEMENT

Sequence of Status Records

Status records are placed in a sequence based upon row number and the type of the diagnostic.

If there are two or more status records, the sequence of the records is determined first by row number. The following rules apply to determining the sequence of errors by row:

- Records that do not correspond to any row appear in front of records that correspond to a particular row, since SQL_NO_ROW_NUMBER is defined to be -1.
- Records for which the row number is unknown appear in front of all other records, since SQL_ROW_NUMBER_UNKNOWN is defined to be -2.

- For all records that pertain to specific rows, records are sorted by the value in the SQL_DIAG_ROW_NUMBER field. All errors and warnings of the first row affected are listed, then all errors and warnings of the next row affected, and so on.

Within each row, or for all those records that do not correspond to a row or for which the row number is unknown, the first record listed is determined using a set of sorting rules. After the first record, the order of the other records affecting a row is undefined. An application cannot assume that errors precede warnings after the first record. Applications should scan the entire diagnostic data structure to obtain complete information on an unsuccessful call to a function.

The following rules are followed to determine the first record within a row. The record with the highest rank is the first record.

- **Errors**. Status records that describe errors have the highest rank. The following rules are followed to sort errors:
  - Records that indicate a transaction failure or possible transaction failure outrank all other records.
  - If two or more records describe the same error condition, then SQLSTATEs defined by the X/Open CLI specification (classes 03 through HZ) outrank ODBC- and driver-defined SQLSTATEs.
- **Implementation-defined No Data values**. Status records that describe DB2 CLI No Data values (class 02) have the second highest rank.
- **Warnings**. Status records that describe warnings (class 01) have the lowest rank. If two or more records describe the same warning condition, then warning SQLSTATEs defined by the X/Open CLI specification outrank ODBC- and driver-defined SQLSTATEs.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA

Diagnostics

`SQLGetDiagField()` does not post error values for itself. It uses the following return values to report the outcome of its own execution:

- SQL_SUCCESS: The function successfully returned diagnostic information.
- SQL_SUCCESS_WITH_INFO: **DiagInfoPtr* was too small to hold the requested diagnostic field so the data in the diagnostic field was truncated.

To determine that a truncation occurred, the application must compare *BufferLength* to the actual number of bytes available, which is written to **StringLengthPtr*.

- SQL_INVALID_HANDLE: The handle indicated by *HandleType* and *Handle* was not a valid handle.
- SQL_ERROR: One of the following occurred:
  - The *DiagIdentifier* argument was not one of the valid values.
  - The *DiagIdentifier* argument was SQL_DIAG_CURSOR_ROW_COUNT, SQL_DIAG_DYNAMIC_FUNCTION, SQL_DIAG_DYNAMIC_FUNCTION_CODE, or SQL_DIAG_ROW_COUNT, but *Handle* was not a statement handle.
  - The *RecNumber* argument was negative or 0 when *DiagIdentifier* indicated a field from a diagnostic record. *RecNumber* is ignored for header fields.
  - The value requested was a character string and *BufferLength* was less than zero.
- SQL_NO_DATA: *RecNumber* was greater than the number of diagnostic records that existed for the handle specified in *Handle*. The function also returns SQL_NO_DATA for any positive *RecNumber* if there are no diagnostic records for *Handle*.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLGetDiagRec - Get Multiple Fields Settings of Diagnostic Record" on page 477

SQLGetDiagRec - Get Multiple Fields Settings of Diagnostic Record

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLGetDiagRec() returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information. Unlike SQLGetDiagField(), which returns one diagnostic field per call, SQLGetDiagRec() returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the error message text.

Syntax

```
SQLRETURN   SQLGetDiagRec    (SQLSMALLINT       HandleType,
                              SQLHANDLE         Handle,
                              SQLSMALLINT       RecNumber,
                              SQLCHAR           *SQLState,
                              SQLINTEGER        *NativeErrorPtr,
                              SQLCHAR           *MessageText,
                              SQLSMALLINT       BufferLength,
                              SQLSMALLINT       *TextLengthPtr);
```

Function Arguments

Table 106. SQLGetDiagRec Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	A handle type identifier that describes the type of handle for which diagnostics are desired. Must be one of the following: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC
SQLHANDLE	*Handle*	input	A handle for the diagnostic data structure, of the type indicated by *HandleType*.
SQLSMALLINT	*RecNumber*	input	Indicates the status record from which the application seeks information. Status records are numbered from 1.
SQLCHAR	*SQLState*	output	Pointer to a buffer in which to return a five-character SQLSTATE code pertaining to the diagnostic record *RecNumber*. The first two characters indicate the class; the next three indicate the subclass.
SQLINTEGER	*NativeErrorPtr*	output	Pointer to a buffer in which to return the native error code, specific to the data source.

Table 106. SQLGetDiagRec Arguments (continued)

Data Type	Argument	Use	Description
SQLCHAR	*MessageText*	output	Pointer to a buffer in which to return the error message text. The fields returned by SQLGetDiagRec() are contained in a text string.
SQLINTEGER	BufferLength	input	Length (in bytes) of the **MessageText* buffer.
SQLSMALLINT	*TextLengthPtr*	output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes required for the null termination character) available to return in **MessageText*. If the number of bytes available to return is greater than *BufferLength*, then the error message text in **MessageText* is truncated to *BufferLength* minus the length of a null termination character.

Usage

An application typically calls SQLGetDiagRec() when a previous call to a DB2 CLI function has returned anything other than SQL_SUCCESS. However, any function can post zero or more errors each time it is called, so an application can call SQLGetDiagRec() after any function call. An application can call SQLGetDiagRec() multiple times to return some or all of the records in the diagnostic data structure.

SQLGetDiagRec() returns a character string containing multiple fields of the diagnostic data structure record. More information about the data returned can be found in "SQLGetDiagField - Get a Field of Diagnostic Data" on page 468.

SQLGetDiagRec() cannot be used to return fields from the header of the diagnostic data structure (the *RecNumber* argument must be greater than 0). The application should call SQLGetDiagField() for this purpose.

SQLGetDiagRec() retrieves only the diagnostic information most recently associated with the handle specified in the *Handle* argument. If the application calls another function, except SQLGetDiagRec() or SQLGetDiagField(), any diagnostic information from the previous calls on the same handle is lost.

An application can scan all diagnostic records by looping, incrementing *RecNumber*, as long as SQLGetDiagRec() returns SQL_SUCCESS. Calls to SQLGetDiagRec() are non-destructive to the header and record fields. The application can call SQLGetDiagRec() again at a later time to retrieve a field from a record, as long as no other function, except SQLGetDiagRec() or SQLGetDiagField(), has been called in the interim. The application can also retrieve a count of the total number of diagnostic records available by calling

SQLGetDiagField() to retrieve the value of the SQL_DIAG_NUMBER field, then call SQLGetDiagRec() that many times.

For a description of the fields of the diagnostic data structure, see "SQLGetDiagField - Get a Field of Diagnostic Data" on page 468.

HandleType Argument

Each handle type can have diagnostic information associated with it. The *HandleType* argument denotes the handle type of *Handle*.

Some header and record fields cannot be returned for all types of handles: environment, connection, statement, and descriptor. Those handles for which a field is not applicable are indicated in "Header Fields" on page 470 and "Record Fields" on page 472 in the description of SQLGetDescField().

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

SQLGetDiagRec() does not post error values for itself. It uses the following return values to report the outcome of its own execution:

- SQL_SUCCESS: The function successfully returned diagnostic information.
- SQL_SUCCESS_WITH_INFO: The **MessageText* buffer was too small to hold the requested diagnostic message. No diagnostic records were generated. To determine that a truncation occurred, the application must compare *BufferLength* to the actual number of bytes available, which is written to *StringLengthPtr.
- SQL_INVALID_HANDLE: The handle indicated by *HandleType* and *Handle* was not a valid handle.
- SQL_ERROR: One of the following occurred:
  - *RecNumber* was negative or 0.
  - *BufferLength* was less than zero.
- SQL_NO_DATA: *RecNumber* was greater than the number of diagnostic records that existed for the handle specified in *Handle*. The function also returns SQL_NO_DATA for any positive *RecNumber* if there are no diagnostic records for *Handle*.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLGetDiagField - Get a Field of Diagnostic Data" on page 468

SQLGetEnvAttr - Retrieve Current Environment Attribute Value

Purpose

Specification:	DB2 CLI 2.1		ISO CLI

SQLGetEnvAttr() returns the current setting for the specified environment attribute.

These options are set using the SQLSetEnvAttr() function.

Syntax

```
SQLRETURN   SQLGetEnvAttr    (
                SQLHENV           EnvironmentHandle, /* henv */
                SQLINTEGER        Attribute,
                SQLPOINTER        ValuePtr,          /* Value */
                SQLINTEGER        BufferLength,
                SQLINTEGER   *FAR StringLengthPtr);  /* StringLength */
```

Function Arguments

Table 107. SQLGetEnvAttr Arguments

Data Type	Argument	Use	Description
SQLHENV	EnvironmentHandle	input	Environment handle.
SQLINTEGER	Attribute	input	Attribute to receive. Refer to "Environment Attributes" on page 682 for the list of environment attributes and their descriptions.
SQLPOINTER	ValuePtr	output	The current value associated with *Attribute*. The type of the value returned depends on *Attribute*.
SQLINTEGER	BufferLength	input	Maximum size of buffer pointed to by *ValuePtr*, if the attribute value is a character string; otherwise, ignored.
SQLINTEGER *	StringLengthPtr	output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes returned for the null-termination character) available to return in *ValuePtr*. If *ValuePtr* is a null pointer, no longth is returned. If the attribute value is a character string, and the number of bytes available to return is greater than or equal to *BufferLength*, the data in *ValuePtr* is truncated to *BufferLength* minus the length of a null-termination character and is null-terminated by DB2 CLI

If *Attribute* does not denote a string, then DB2 CLI ignores *BufferLength* and does not set *StringLengthPtr*.

SQLGetEnvAttr

Usage

SQLGetEnvAttr() can be called at any time between the allocation and freeing of the environment handle. It obtains the current value of the environment attribute.

For a list of valid environment attributes, refer to "Environment Attributes" on page 682.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 108. SQLGetEnvAttr SQLSTATEs

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY092	Option type out of range.	An invalid *Attribute* value was specified.

Restrictions

None.

Example

```
/* From CLI sample getattrs.c */
/* ... */
    rc = SQLGetEnvAttr( henv, SQL_ATTR_OUTPUT_NTS, &output_nts, 0, NULL ) ;
    CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc ) ;
    printf( "Null Termination of Output strings is: " ) ;
    if ( output_nts == SQL_TRUE ) printf( "True\n" ) ;
    else printf( "False\n" ) ;
```

References

- "SQLSetEnvAttr - Set Environment Attribute" on page 682

SQLGetFunctions - Get Functions

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	ISO CLI

SQLGetFunctions() to query whether a specific function is supported. This allows applications to adapt to varying levels of support when connecting to different database servers.

A connection to a database server must exist before calling this function.

Syntax

```
SQLRETURN   SQLGetFunctions  (
                SQLHDBC           ConnectionHandle,  /* hdbc */
                SQLUSMALLINT      FunctionId,        /* fFunction */
                SQLUSMALLINT *FAR SupportedPtr);     /* pfExists */
```

Function Arguments

Table 109. SQLGetFunctions Arguments

Data Type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Database connection handle.
SQLUSMALLINT	*FunctionId*	input	The function being queried. Valid *FunctionId* values are shown in Figure 17 on page 484
SQLUSMALLINT *	*SupportedPtr*	output	Pointer to location where this function will return SQL_TRUE or SQL_FALSE depending on whether the function being queried is supported.

Usage

Figure 17 on page 484 shows the valid values for the *FunctionId* argument and whether the corresponding function is supported. (This list was generated using the sample application getfuncs.c.)

If *FunctionId* is set to SQL_API_ALL_FUNCTIONS, then *SupportedPtr* must point to an SQLSMALLINT array of 100 elements. The array is indexed by the *FunctionId* values used to identify many of the functions. Some elements of the array are unused and reserved. Since some *FunctionId* values are greater than 100, the array method can not be used to obtain a list of functions. The SQLGetFunction() call must be explicitly issued for all *FunctionId* values equal to or above 100. The complete set of *FunctionId* values is defined in sqlcli1.h.

SQLGetFunctions

```
------------------------------------------
Connected to Server: SAMPLE
 Database Name: SAMPLE
 Instance Name: db2inst1
     DBMS Name: DB2/6000
  DBMS Version: 05.00.0000
------------------------------------------
SQLALLOCCONNECT      is supported   SQLALLOCENV          is supported
SQLALLOCHANDLE       is supported   SQLALLOCSTMT         is supported
SQLBINDCOL           is supported   SQLBINDFILETOCOL     is supported
SQLBINDFILETOPARAM   is supported   SQLBINDPARAM         is supported
SQLBINDPARAMETER     is supported   SQLBROWSECONNECT     is supported
SQLCANCEL            is supported   SQLCLOSECURSOR       is supported
SQLCOLATTRIBUTE      is supported   SQLCOLATTRIBUTES     is supported
SQLCOLUMNPRIVILEGES  is supported   SQLCOLUMNS           is supported
SQLCONNECT           is supported   SQLCOPYDESC          is supported
SQLDATASOURCES       is supported   SQLDESCRIBECOL       is supported
SQLDESCRIBEPARAM     is supported   SQLDISCONNECT        is supported
SQLDRIVERCONNECT     is supported   SQLENDTRAN           is supported
SQLERROR             is supported   SQLEXECDIRECT        is supported
SQLEXECUTE           is supported   SQLEXTENDEDFETCH     is supported
SQLFETCH             is supported   SQLFETCHSCROLL       is supported
SQLFOREIGNKEYS       is supported   SQLFREECONNECT       is supported
SQLFREEENV           is supported   SQLFREEHANDLE        is supported
SQLFREESTMT          is supported   SQLGETCONNECTATTR    is supported
SQLGETCONNECTOPTION  is supported   SQLGETCURSORNAME     is supported
SQLGETDATA           is supported   SQLGETDESCFIELD      is supported
SQLGETDESCREC        is supported   SQLGETDIAGFIELD      is supported
SQLGETDIAGREC        is supported   SQLGETENVATTR        is supported
SQLGETFUNCTIONS      is supported   SQLGETINFO           is supported
SQLGETLENGTH         is supported   SQLGETPOSITION       is supported
SQLGETSQLCA          is supported   SQLGETSTMTATTR       is supported
SQLGETSTMTOPTION     is supported   SQLGETSUBSTRING      is supported
SQLGETTYPEINFO       is supported   SQLMORERESULTS       is supported
SQLNATIVESQL         is supported   SQLNUMPARAMS         is supported
SQLNUMRESULTCOLS     is supported   SQLPARAMDATA         is supported
SQLPARAMOPTIONS      is supported   SQLPREPARE           is supported
SQLPRIMARYKEYS       is supported   SQLPROCEDURECOLUMNS  is supported
SQLPROCEDURES        is supported   SQLPUTDATA           is supported
SQLROWCOUNT          is supported   SQLSETCOLATTRIBUTES  is supported
SQLSETCONNECTATTR    is supported   SQLSETCONNECTION     is supported
SQLSETCONNECTOPTION  is supported   SQLSETCURSORNAME     is supported
SQLSETDESCFIELD      is supported   SQLSETDESCREC        is supported
SQLSETENVATTR        is supported   SQLSETPARAM          is supported
SQLSETPOS            is supported   SQLSETSCROLLOPTIONS  is supported
SQLSETSTMTATTR       is supported   SQLSETSTMTOPTION     is supported
SQLSPECIALCOLUMNS    is supported   SQLSTATISTICS        is supported
SQLTABLEPRIVILEGES   is supported   SQLTABLES            is supported
SQLTRANSACT          is supported
```

Figure 17. Supported Functions list (output from getfuncs.c).

Note: The LOB support functions (SQLGetLength(), SQLGetPosition(), SQLGetSubString(), SQLBindFileToCol(), SQLBindFileToCol()) are not

supported when connected to DB2 for common server prior to Version 2.1 or other IBM RDBMSs that do not support LOB data types.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 110. SQLGetFunctions SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	SQLGetFunctions() was called before a database connection was established.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Authorization

None.

Example

The following example generates the listing shown in Figure 17 on page 484 for any data source.

```
/* From CLI sample getfuncs.c */
/* ... */

typedef struct {
  SQLUSMALLINT id ;
  char * name ;
} functionInfo ;

functionInfo functions[] = {
  { SQL_API_SQLALLOCCONNECT,     "SQLALLOCCONNECT"     },
  { SQL_API_SQLALLOCENV,         "SQLALLOCENV"         },
  { SQL_API_SQLALLOCHANDLE,      "SQLALLOCHANDLE"      },
  { SQL_API_SQLALLOCSTMT,        "SQLALLOCSTMT"        },
  { SQL_API_SQLBINDCOL,          "SQLBINDCOL"          },
  { SQL_API_SQLBINDFILETOCOL,    "SQLBINDFILETOCOL"    },
```

```
{ SQL_API_SQLBINDFILETOPARAM,  "SQLBINDFILETOPARAM"  },
{ SQL_API_SQLBINDPARAM,        "SQLBINDPARAM"        },
{ SQL_API_SQLBINDPARAMETER,    "SQLBINDPARAMETER"    },
{ SQL_API_SQLBROWSECONNECT,    "SQLBROWSECONNECT"    },
{ SQL_API_SQLCANCEL,           "SQLCANCEL"           },
{ SQL_API_SQLCLOSECURSOR,      "SQLCLOSECURSOR"      },
{ SQL_API_SQLCOLATTRIBUTE,     "SQLCOLATTRIBUTE"     },
{ SQL_API_SQLCOLATTRIBUTES,    "SQLCOLATTRIBUTES"    },
{ SQL_API_SQLCOLUMNPRIVILEGES, "SQLCOLUMNPRIVILEGES" },
{ SQL_API_SQLCOLUMNS,          "SQLCOLUMNS"          },
{ SQL_API_SQLCONNECT,          "SQLCONNECT"          },
{ SQL_API_SQLCOPYDESC,         "SQLCOPYDESC"         },
{ SQL_API_SQLDATASOURCES,      "SQLDATASOURCES"      },
{ SQL_API_SQLDESCRIBECOL,      "SQLDESCRIBECOL"      },
{ SQL_API_SQLDESCRIBEPARAM,    "SQLDESCRIBEPARAM"    },
{ SQL_API_SQLDISCONNECT,       "SQLDISCONNECT"       },
{ SQL_API_SQLDRIVERCONNECT,    "SQLDRIVERCONNECT"    },
{ SQL_API_SQLENDTRAN,          "SQLENDTRAN"          },
{ SQL_API_SQLERROR,            "SQLERROR"            },
{ SQL_API_SQLEXECDIRECT,       "SQLEXECDIRECT"       },
{ SQL_API_SQLEXECUTE,          "SQLEXECUTE"          },
{ SQL_API_SQLEXTENDEDFETCH,    "SQLEXTENDEDFETCH"    },
{ SQL_API_SQLFETCH,            "SQLFETCH"            },
{ SQL_API_SQLFETCHSCROLL,      "SQLFETCHSCROLL"      },
{ SQL_API_SQLFOREIGNKEYS,      "SQLFOREIGNKEYS"      },
{ SQL_API_SQLFREECONNECT,      "SQLFREECONNECT"      },
{ SQL_API_SQLFREEENV,          "SQLFREEENV"          },
{ SQL_API_SQLFREEHANDLE,       "SQLFREEHANDLE"       },
{ SQL_API_SQLFREESTMT,         "SQLFREESTMT"         },
{ SQL_API_SQLGETCONNECTATTR,   "SQLGETCONNECTATTR"   },
{ SQL_API_SQLGETCONNECTOPTION, "SQLGETCONNECTOPTION" },
{ SQL_API_SQLGETCURSORNAME,    "SQLGETCURSORNAME"    },
{ SQL_API_SQLGETDATA,          "SQLGETDATA"          },
{ SQL_API_SQLGETDESCFIELD,     "SQLGETDESCFIELD"     },
{ SQL_API_SQLGETDESCREC,       "SQLGETDESCREC"       },
{ SQL_API_SQLGETDIAGFIELD,     "SQLGETDIAGFIELD"     },
{ SQL_API_SQLGETDIAGREC,       "SQLGETDIAGREC"       },
{ SQL_API_SQLGETENVATTR,       "SQLGETENVATTR"       },
{ SQL_API_SQLGETFUNCTIONS,     "SQLGETFUNCTIONS"     },
{ SQL_API_SQLGETINFO,          "SQLGETINFO"          },
{ SQL_API_SQLGETLENGTH,        "SQLGETLENGTH"        },
{ SQL_API_SQLGETPOSITION,      "SQLGETPOSITION"      },
{ SQL_API_SQLGETSQLCA,         "SQLGETSQLCA"         },
{ SQL_API_SQLGETSTMTATTR,      "SQLGETSTMTATTR"      },
{ SQL_API_SQLGETSTMTOPTION,    "SQLGETSTMTOPTION"    },
{ SQL_API_SQLGETSUBSTRING,     "SQLGETSUBSTRING"     },
{ SQL_API_SQLGETTYPEINFO,      "SQLGETTYPEINFO"      },
{ SQL_API_SQLMORERESULTS,      "SQLMORERESULTS"      },
{ SQL_API_SQLNATIVESQL,        "SQLNATIVESQL"        },
{ SQL_API_SQLNUMPARAMS,        "SQLNUMPARAMS"        },
{ SQL_API_SQLNUMRESULTCOLS,    "SQLNUMRESULTCOLS"    },
{ SQL_API_SQLPARAMDATA,        "SQLPARAMDATA"        },
{ SQL_API_SQLPARAMOPTIONS,     "SQLPARAMOPTIONS"     },
{ SQL_API_SQLPREPARE,          "SQLPREPARE"          },
{ SQL_API_SQLPRIMARYKEYS,      "SQLPRIMARYKEYS"      },
```

```
  { SQL_API_SQLPROCEDURECOLUMNS, "SQLPROCEDURECOLUMNS" },
  { SQL_API_SQLPROCEDURES,       "SQLPROCEDURES"       },
  { SQL_API_SQLPUTDATA,          "SQLPUTDATA"          },
  { SQL_API_SQLROWCOUNT,         "SQLROWCOUNT"         },
  { SQL_API_SQLSETCOLATTRIBUTES, "SQLSETCOLATTRIBUTES" },
  { SQL_API_SQLSETCONNECTATTR,   "SQLSETCONNECTATTR"   },
  { SQL_API_SQLSETCONNECTION,    "SQLSETCONNECTION"    },
  { SQL_API_SQLSETCONNECTOPTION, "SQLSETCONNECTOPTION" },
  { SQL_API_SQLSETCURSORNAME,    "SQLSETCURSORNAME"    },
  { SQL_API_SQLSETDESCFIELD,     "SQLSETDESCFIELD"     },
  { SQL_API_SQLSETDESCREC,       "SQLSETDESCREC"       },
  { SQL_API_SQLSETENVATTR,       "SQLSETENVATTR"       },
  { SQL_API_SQLSETPARAM,         "SQLSETPARAM"         },
  { SQL_API_SQLSETPOS,           "SQLSETPOS"           },
  { SQL_API_SQLSETSCROLLOPTIONS, "SQLSETSCROLLOPTIONS" },
  { SQL_API_SQLSETSTMTATTR,      "SQLSETSTMTATTR"      },
  { SQL_API_SQLSETSTMTOPTION,    "SQLSETSTMTOPTION"    },
  { SQL_API_SQLSPECIALCOLUMNS,   "SQLSPECIALCOLUMNS"   },
  { SQL_API_SQLSTATISTICS,       "SQLSTATISTICS"       },
  { SQL_API_SQLTABLEPRIVILEGES,  "SQLTABLEPRIVILEGES"  },
  { SQL_API_SQLTABLES,           "SQLTABLES"           },
  { SQL_API_SQLTRANSACT,         "SQLTRANSACT"         },
  { 0,                           ( char * ) 0          }
} ;

/* ... */

    func_pos = 0 ;
    side = 0 ;
    *b_line = '\0' ;
    *e_line = '\0' ;

    while ( functions[func_pos].id != 0 ) {
       SQLGetFunctions( hdbc,
                        functions[func_pos].id,
                        &supported
                      ) ;
       if ( supported )
          printf( "%s%-20s is supported%s",
                  b_line,
                  functions[func_pos].name,
                  e_line
                ) ;
       else
          printf( "%s%-20s is not supported%s",
                  b_line,
                  functions[func_pos].name,
                  e_line
                ) ;
       if ( side ) {
          *b_line = '\0' ;
          *e_line = '\0' ;
          side = 0 ;
       }
       else {
```

```
            strcpy( b_line, "    " ) ;
            strcpy( e_line, "\n" ) ;
            side = 1 ;
         }
         func_pos++ ;
      }
```

References

None.

SQLGetInfo - Get General Information

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLGetInfo() returns general information, (including supported data conversions) about the DBMS that the application is currently connected to.

Syntax

```
SQLRETURN   SQLGetInfo       (
                SQLHDBC           ConnectionHandle,  /* hdbc */
                SQLUSMALLINT      InfoType,          /* fInfoType */
                SQLPOINTER        InfoValuePtr,      /* rgbInfoValue */
                SQLSMALLINT       BufferLength,      /* cbInfoValueMax */
                SQLSMALLINT  *FAR StringLengthPtr);  /* pcbInfoValue */
```

Function Arguments

Table 111. SQLGetInfo Arguments

Data Type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Database connection handle
SQLUSMALLINT	*InfoType*	input	The type of information desired. The argument must be one of the values in the first column of the tables in “Data Types and Data Conversion” on page 28.
SQLPOINTER	*InfoValuePtr*	output (also input)	Pointer to buffer where this function will store the desired information. Depending on the type of information being retrieved, 5 types of information can be returned: • 16 bit integer value • 32 bit integer value • 32 bit binary value • 32 bit mask • null-terminated character string
SQLSMALLINT	*BufferLength*	input	Maximum length of the buffer pointed by *InfoValuePtr* pointer.

Table 111. SQLGetInfo Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT *	*StringLengthPtr*	output	Pointer to location where this function will return the total number of bytes available to return the desired information. In the case of string output, this size does not include the null terminating character. If the value in the location pointed to by *StringLengthPtr* is greater than the size of the *InfoValuePtr* buffer as specified in *BufferLength*, then the string output information would be truncated to *BufferLength* - 1 bytes and the function would return with SQL_SUCCESS_WITH_INFO.

Usage

Refer to “Information Returned By SQLGetInfo” for a list of the possible values of *InfoType* and a description of the information that `SQLGetInfo()` would return for that value.

A number of information types were renamed for DB2 CLI version 5. See “Changes to the InfoTypes in SQLGetInfo()” on page 772 for the list. “Information Returned By SQLGetInfo” lists both the old value and the new value.

Information Returned By SQLGetInfo

Note: DB2 CLI returns a value for each *InfoType* in this table. If the *InfoType* does not apply or is not supported, the result is dependent on the return type. If the return type is a:

- Character string containing ’Y’ or ’N’, ″N″ is returned.
- Character string containing a value other than just ’Y’ or ’N’, an empty string is returned.
- 16-bit integer, 0 (zero).
- 32-bit integer, 0 (zero).
- 32-bit mask, 0 (zero).

SQL_ACCESSIBLE_PROCEDURES (string)
A character string of ″Y″ indicates that the user can execute all procedures returned by the function `SQLProcedures()`. ″N″ indicates there may be procedures returned that the user cannot execute.

SQL_ACCESSIBLE_TABLES (string)
A character string of ″Y″ indicates that the user is guaranteed SELECT

privilege to all tables returned by the function SQLTables(). "N" indicates that there may be tables returned that the user cannot access.

SQL_ACTIVE_ENVIRONMENTS (16-bit integer)
This *InfoType* has been replaced with SQL_MAX_CONCURRENT_ACTIVITIES.

The maximum number of active environments that the DB2 CLI driver can support. If there is no specified limit or the limit is unknown, this value is set to zero.

SQL_AGGREGATE_FUNCTIONS (32-bit mask)
A bitmask enumerating support for aggregation functions:

- SQL_AF_ALL
- SQL_AF_AVG
- SQL_AF_COUNT
- SQL_AF_DISTINCT
- SQL_AF_MAX
- SQL_AF_MIN
- SQL_AF_SUM

SQL_ALTER_DOMAIN (32-bit mask)
DB2 CLI returns 0 indicating that the ALTER DOMAIN statement is not supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_AD_ADD_CONSTRAINT_DEFERRABLE
- SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE
- SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED
- SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_AD_ADD_DOMAIN_CONSTRAINT
- SQL_AD_ADD_DOMAIN_DEFAULT
- SQL_AD_CONSTRAINT_NAME_DEFINITION
- SQL_AD_DROP_DOMAIN_CONSTRAINT
- SQL_AD_DROP_DOMAIN_DEFAULT

SQL_ACTIVE_CONNECTIONS (16-bit integer)
This *InfoType* has been replaced with SQL_MAX_DRIVER_CONNECTIONS.

The maximum number of active connections supported per application.

Zero is returned, indicating that the limit is dependent on system resources.

The MAXCONN keyword in the db2cli.ini initialization file or the SQL_ATTR_MAX_CONNECTIONS environment/connection option can be used to impose a limit on the number of connections. This limit is returned if it is set to any value other than zero.

SQL_ACTIVE_STATEMENTS (16-bit integer)
This *InfoType* has been replaced with SQL_MAX_CONCURRENT_ACTIVITIES.

The maximum number of active statements per connection.

Zero is returned, indicating that the limit is dependent on database system and DB2 CLI resources, and limits.

SQL_ALTER_TABLE (32-bit mask)
Indicates which clauses in the ALTER TABLE statement are supported by the DBMS.

- SQL_AT_ADD_COLUMN_COLLATION
- SQL_AT_ADD_COLUMN_DEFAULT
- SQL_AT_ADD_COLUMN_SINGLE
- SQL_AT_ADD_CONSTRAINT
- SQL_AT_ADD_TABLE_CONSTRAINT
- SQL_AT_CONSTRAINT_NAME_DEFINITION
- SQL_AT_DROP_COLUMN_CASCADE
- SQL_AT_DROP_COLUMN_DEFAULT
- SQL_AT_DROP_COLUMN_RESTRICT
- SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE
- SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT
- SQL_AT_SET_COLUMN_DEFAULT
- SQL_AT_CONSTRAINT_INITIALLY_DEFERRED
- SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_AT_CONSTRAINT_DEFERRABLE
- SQL_AT_CONSTRAINT_NON_DEFERRABLE

SQL_ASYNC_MODE (32-bit unsigned integer)
Indicates the level of asynchronous support:

- SQL_AM_CONNECTION, connection level asynchronous execution is supported. Either all statement handles associated with a given connection handle are in asynchronous mode, or all are in synchronous mode. A statement handle on a connection cannot be in asynchronous mode while another statement handle on the same connection is in synchronous mode, and vice versa.
- SQL_AM_STATEMENT, statement level asynchronous execution is supported. Some statement handles associated with a connection

handle can be in asynchronous mode, while other statement handles on the same connection are in synchronous mode.

- SQL_AM_NONE, asynchronous mode is not supported.

  This value is also returned if the DB2 CLI/ODBC configuration keyword ASYNCENABLE is set to disable asynchronous execution. See "Asynchronous Execution of CLI" on page 138 for more information.

SQL_BATCH_ROW_COUNT (32-bit mask)

Indicates how row counts are dealt with. DB2 CLI always returns SQL_BRC_ROLLED_UP indicating that row counts for consecutive INSERT, DELETE, or UPDATE statements are rolled up into one.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_BRC_PROCEDURES
- SQL_BRC_EXPLICIT

SQL_BATCH_SUPPORT (32-bit mask)

Indicates which level of batches are supported:

- SQL_BS_SELECT_EXPLICIT, supports explicit batches that can have result-set generating statements.
- SQL_BS_ROW_COUNT_EXPLICIT, supports explicit batches that can have row-count generating statements.
- SQL_BS_SELECT_PROC, supports explicit procedures that can have result-set generating statements.
- SQL_BS_ROW_COUNT_PROC, supports explicit procedures that can have row-count generating statements.

SQL_BOOKMARK_PERSISTENCE (32-bit mask)

Indicates when bookmarks remain valid after an operation:

- SQL_BP_CLOSE, bookmarks are valid after an application calls `SQLFreeStmt()` with the SQL_CLOSE option, `SQLCloseCursor()` to close the cursor associated with a statement.
- SQL_BP_DELETE, the bookmark for a row is valid after that row has been deleted.
- SQL_BP_DROP, bookmarks are valid after an application calls `SQLFreeHandle()` with a *HandleType* of SQL_HANDLE_STMT to drop a statement.
- SQL_BP_TRANSACTION, bookmarks are valid after an application commits or rolls back a transaction.
- SQL_BP_UPDATE, the bookmark for a row is valid after any column in that row has been updated, including key columns.

- SQL_BP_OTHER_HSTMT, a bookmark associated with one statement can be used with another statement. Unless SQL_BP_CLOSE or SQL_BP_DROP is specified, the cursor on the first statement must be open.
- SQL_BP_DROP is specified, the cursor on the first statement must be open.

SQL_CATALOG_LOCATION (16-bit integer)
A 16-bit integer value indicated the position of the qualifier in a qualified table name. DB2 CLI always returns SQL_CL_START for this information type. ODBC also defines the value SQL_CL_END which is not returned by DB2 CLI.

In previous versions of DB2 CLI this *InfoType* was SQL_QUALIFIER_LOCATION.

SQL_CATALOG_NAME (string)
A character string of ″Y″ indicates that the server supports catalog names. ″N″ indicates that catalog names are not supported.

SQL_CATALOG_NAME_SEPARATOR (string)
The character(s) used as a separator between a catalog name and the qualified name element that follows it.

In previous versions of DB2 CLI this *InfoType* was SQL_QUALIFIER_NAME_SEPARATOR.

SQL_CATALOG_TERM (string)
The database vendor's terminology for a qualifier

The name that the vendor uses for the high order part of a three part name.

Since DB2 CLI does not support three part names, a zero-length string is returned.

In previous versions of DB2 CLI this *InfoType* was SQL_QUALIFIER_TERM.

SQL_CATALOG_USAGE (32-bit mask)
This is similar to SQL_OWNER_USAGE except that this is used for catalog.

In previous versions of DB2 CLI this *InfoType* was SQL_QUALIFIER_USAGE.

SQL_COLLATION_SEQ (string)
The name of the collation sequence. This is a character string that indicates the name of the default collation for the default character set for this server (for example ISO 8859-1 or EBCDIC). If this is unknown, an empty string will be returned.

SQL_COLUMN_ALIAS (string)
Returns ″Y″ if column aliases are supported, or ″N″ if they are not.

SQL_CONCAT_NULL_BEHAVIOR (16-bit integer)
Indicates how the concatenation of NULL valued character data type columns with non-NULL valued character data type columns is handled.

- SQL_CB_NULL - indicates the result is a NULL value (this is the case for IBM RDBMs).
- SQL_CB_NON_NULL - indicates the result is a concatenation of non-NULL column values.

SQL_CONVERT_BIGINT
SQL_CONVERT_BINARY
SQL_CONVERT_BIT
SQL_CONVERT_CHAR
SQL_CONVERT_DATE
SQL_CONVERT_DECIMAL
SQL_CONVERT_DOUBLE
SQL_CONVERT_FLOAT
SQL_CONVERT_INTEGER
SQL_CONVERT_INTERVAL_YEAR_MONTH
SQL_CONVERT_INTERVAL_DAY_TIME
SQL_CONVERT_LONGVARBINARY
SQL_CONVERT_LONGVARCHAR
SQL_CONVERT_NUMERIC
SQL_CONVERT_REAL
SQL_CONVERT_SMALLINT
SQL_CONVERT_TIME
SQL_CONVERT_TIMESTAMP
SQL_CONVERT_TINYINT
SQL_CONVERT_VARBINARY
SQL_CONVERT_VARCHAR

(all above are 32-bit masks)
Indicates the conversions supported by the data source with the CONVERT scalar function for data of the type named in the *InfoType*. If the bitmask equals zero, the data source does not support any conversions for the data of the named type, including conversions to the same data type.

For example, to find out if a data source supports the conversion of SQL_INTEGER data to the SQL_DECIMAL data type, an application calls `SQLGetInfo()` with *InfoType* of SQL_CONVERT_INTEGER. The application then ANDs the returned bitmask with SQL_CVT_DECIMAL. If the resulting value is nonzero then the conversion is supported.

The following bitmasks are used to determine which conversions are supported:

- SQL_CVT_BIGINT
- SQL_CVT_BINARY
- SQL_CVT_BIT
- SQL_CVT_CHAR
- SQL_CVT_DATE
- SQL_CVT_DECIMAL
- SQL_CVT_DOUBLE
- SQL_CVT_FLOAT
- SQL_CVT_INTEGER
- SQL_CVT_INTERVAL_YEAR_MONTH
- SQL_CVT_INTERVAL_DAY_TIME
- SQL_CVT_LONGVARBINARY
- SQL_CVT_LONGVARCHAR
- SQL_CVT_NUMERIC
- SQL_CVT_REAL
- SQL_CVT_SMALLINT
- SQL_CVT_TIME
- SQL_CVT_TIMESTAMP
- SQL_CVT_TINYINT
- SQL_CVT_VARBINARY
- SQL_CVT_VARCHAR

SQL_CONVERT_FUNCTIONS (32-bit mask)

Indicates the scalar conversion functions supported by the driver and associated data source.

DB2 CLI Version 2.1.1 and later supports ODBC scalar conversions between char variables (CHAR, VARCHAR, LONG VARCHAR and CLOB) and DOUBLE (or FLOAT).

- SQL_FN_CVT_CONVERT - used to determine which conversion functions are supported.

SQL_CORRELATION_NAME (16-bit integer)

Indicates the degree of correlation name support by the server:

- SQL_CN_ANY, supported and can be any valid user-defined name.
- SQL_CN_NONE, correlation name not supported.
- SQL_CN_DIFFERENT, correlation name supported but it must be different than the name of the table that it represent.

SQL_CREATE_ASSERTION (32-bit mask)
Indicates which clauses in the CREATE ASSERTION statement are supported by the DBMS. DB2 CLI always returns zero; the CREATE ASSERTION statement is not supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_CA_CREATE_ASSERTION
- SQL_CA_CONSTRAINT_INITIALLY_DEFERRED
- SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_CA_CONSTRAINT_DEFERRABLE
- SQL_CA_CONSTRAINT_NON_DEFERRABLE

SQL_CREATE_CHARACTER_SET (32-bit mask)
Indicates which clauses in the CREATE CHARACTER SET statement are supported by the DBMS. DB2 CLI always returns zero; the CREATE CHARACTER SET statement is not supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_CCS_CREATE_CHARACTER_SET
- SQL_CCS_COLLATE_CLAUSE
- SQL_CCS_LIMITED_COLLATION

SQL_CREATE_COLLATION (32-bit mask)
Indicates which clauses in the CREATE COLATION statement are supported by the DBMS. DB2 CLI always returns zero; the CREATE COLLATION statement is not supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_CCOL_CREATE_COLLATION

SQL_CREATE_DOMAIN (32-bit mask)
Indicates which clauses in the CREATE DOMAIN statement are supported by the DBMS. DB2 CLI always returns zero; the CREATE DOMAIN statement is not supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_CDO_CREATE_DOMAIN
- SQL_CDO_CONSTRAINT_NAME_DEFINITION
- SQL_CDO_DEFAULT
- SQL_CDO_CONSTRAINT
- SQL_CDO_COLLATION
- SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED

- SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_CDO_CONSTRAINT_DEFERRABLE
- SQL_CDO_CONSTRAINT_NON_DEFERRABLE

SQL_CREATE_SCHEMA (32-bit mask)
Indicates which clauses in the CREATE SCHEMA statement are supported by the DBMS:

- SQL_CS_CREATE_SCHEMA
- SQL_CS_AUTHORIZATION
- SQL_CS_DEFAULT_CHARACTER_SET

SQL_CREATE_TABLE (32-bit mask)
Indicates which clauses in the CREATE TABLE statement are supported by the DBMS.

The following bitmasks are used to determine which clauses are supported:

- SQL_CT_CREATE_TABLE
- SQL_CT_TABLE_CONSTRAINT
- SQL_CT_CONSTRAINT_NAME_DEFINITION

The following bits specify the ability to create temporary tables:

- SQL_CT_COMMIT_PRESERVE, deleted rows are preserved on commit.
- SQL_CT_COMMIT_DELETE, deleted rows are deleted on commit.
- SQL_CT_GLOBAL_TEMPORARY, global temporary tables can be created.
- SQL_CT_LOCAL_TEMPORARY, local temporary tables can be created.

The following bits specify the ability to create column constraints:

- SQL_CT_COLUMN_CONSTRAINT, specifying column constraints is supported.
- SQL_CT_COLUMN_DEFAULT, specifying column defaults is supported.
- SQL_CT_COLUMN_COLLATION, specifying column collation is supported.

The following bits specify the supported constraint attributes if specifying column or table constraints is supported:

- SQL_CT_CONSTRAINT_INITIALLY_DEFERRED
- SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_CT_CONSTRAINT_DEFERRABLE

- SQL_CT_CONSTRAINT_NON_DEFERRABLE

SQL_CREATE_TRANSLATION (32-bit mask)
Indicates which clauses in the CREATE TRANSLATION statement are supported by the DBMS. DB2 CLI always returns zero; the CREATE TRANSLATION statement is not supported.

ODBC also defines the following value that is not returned by DB2 CLI:

- SQL_CTR_CREATE_TRANSLATION

SQL_CREATE_VIEW (32-bit mask)
Indicates which clauses in the CREATE VIEW statement are supported by the DBMS:

- SQL_CV_CREATE_VIEW
- SQL_CV_CHECK_OPTION
- SQL_CV_CASCADED
- SQL_CV_LOCAL

A return value of 0 means that the CREATE VIEW statement is not supported.

SQL_CURSOR_CLOSE_BEHAVIOR (32-bit unsigned integer)
Indicates whether or not locks are released when the cursor is closed. The possible values are:

- SQL_CC_NO_RELEASE: locks are not released when the cursor on this statement handle is closed. This is the default.
- SQL_CC_RELEASE: locks are released when the cursor on this statement handle is closed.

Typically cursors are explicitly closed when the function `SQLFreeStmt()` is called with the SQL_CLOSE option or `SQLFreeHandle()` is called with *HandleType* set to SQL_HANDLE_STMT. In addition, the end of the transaction (when a commit or rollback is issued) may also cause the closing of the cursor (depending on the WITH HOLD attribute currently in use).

SQL_CURSOR_COMMIT_BEHAVIOR (16-bit integer)
Indicates how a COMMIT operation affects cursors. A value of:

- SQL_CB_DELETE, destroy cursors and drops access plans for dynamic SQL statements.
- SQL_CB_CLOSE, destroy cursors, but retains access plans for dynamic SQL statements (including non-query statements)
- SQL_CB_PRESERVE, retains cursors and access plans for dynamic statements (including non-query statements). Applications can

continue to fetch data, or close the cursor and re-execute the query without re-preparing the statement.

Note: After COMMIT, a FETCH must be issued to reposition the cursor before actions such as positioned updates or deletes can be taken.

SQL_CURSOR_ROLLBACK_BEHAVIOR (16-bit integer)
Indicates how a ROLLBACK operation affects cursors. A value of:

- SQL_CB_DELETE, destroy cursors and drops access plans for dynamic SQL statements.
- SQL_CB_CLOSE, destroy cursors, but retains access plans for dynamic SQL statements (including non-query statements)
- SQL_CB_PRESERVE, retains cursors and access plans for dynamic statements (including non-query statements). Applications can continue to fetch data, or close the cursor and re-execute the query without re-preparing the statement.

Note: DB2 servers do not have the SQL_CB_PRESERVE property.

SQL_CURSOR_SENSITIVITY (32-bit unsigned integer)
Indicates support for cursor sensitivity:

- SQL_INSENSITIVE, all cursors on the statement handle show the result set without reflecting any changes made to it by any other cursor within the same transaction.
- SQL_UNSPECIFIED, tt is unspecified whether cursors on the statement handle make visible the changes made to a result set by another cursor within the same transaction. Cursors on the statement handle may make visible none, some, or all such changes.
- SQL_SENSITIVE, cursors are sensitive to changes made by other cursors within the same transaction.

SQL_DATA_SOURCE_NAME (string)
The name used as data source on the input to `SQLConnect()`, or the DSN keyword value in the `SQLDriverConnect()` connection string.

SQL_DATA_SOURCE_READ_ONLY (string)
A character string of ″Y″ indicates that the database is set to READ ONLY mode, ″N″ indicates that is not set to READ ONLY mode.

SQL_DATABASE_NAME (string)
The name of the current database in use

Note: also returned by SELECT CURRENT SERVER on IBM DBMS's.

SQL_DATETIME_LITERALS (32-bit unsigned integer)
Indicates the datetime literals that are supported by the DBMS. DB2 CLI always returns zero; datetime literals are not supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_DL_SQL92_DATE
- SQL_DL_SQL92_TIME
- SQL_DL_SQL92_TIMESTAMP
- SQL_DL_SQL92_INTERVAL_YEAR
- SQL_DL_SQL92_INTERVAL_MONTH
- SQL_DL_SQL92_INTERVAL_DAY
- SQL_DL_SQL92_INTERVAL_HOUR
- SQL_DL_SQL92_INTERVAL_MINUTE
- SQL_DL_SQL92_INTERVAL_SECOND
- SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH
- SQL_DL_SQL92_INTERVAL_DAY_TO_HOUR
- SQL_DL_SQL92_INTERVAL_DAY_TO_MINUTE
- SQL_DL_SQL92_INTERVAL_DAY_TO_SECOND
- SQL_DL_SQL92_INTERVAL_HOUR_TO_MINUTE
- SQL_DL_SQL92_INTERVAL_HOUR_TO_SECOND
- SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND

SQL_DBMS_NAME (string)
The name of the DBMS product being accessed

For example:

- ″DB2/6000″
- ″DB2/2″

SQL_DBMS_VER (string)
The Version of the DBMS product accessed. A string of the form 'mm.vv.rrrr' where mm is the major version, vv is the minor version and rrrr is the release. For example, ″0r.01.0000″ translates to major version r, minor version 1, release 0.

SQL_DDL_INDEX (32-bit unsigned integer)
Indicates support for the creation and dropping of indexes:

- SQL_DI_CREATE_INDEX
- SQL_DI_DROP_INDEX

SQL_DEFAULT_TXN_ISOLATION (32-bit mask)
The default transaction isolation level supported

One of the following masks are returned:

- SQL_TXN_READ_UNCOMMITTED = Changes are immediately perceived by all transactions (dirty read, non-repeatable read, and phantoms are possible).

  This is equivalent to IBM's UR level.
- SQL_TXN_READ_COMMITTED = Row read by transaction 1 can be altered and committed by transaction 2 (non-repeatable read and phantoms are possible)

  This is equivalent to IBM's CS level.
- SQL_TXN_REPEATABLE_READ = A transaction can add or remove rows matching the search condition or a pending transaction (repeatable read, but phantoms are possible)

  This is equivalent to IBM's RS level.
- SQL_TXN_SERIALIZABLE = Data affected by pending transaction is not available to other transactions (repeatable read, phantoms are not possible)

  This is equivalent to IBM's RR level.
- SQL_TXN_VERSIONING = Not applicable to IBM DBMSs.
- SQL_TXN_NOCOMMIT = Any chnages are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed.

  This is a DB2 Universal Database for AS/400 isolation level.

In IBM terminology,

- SQL_TXN_READ_UNCOMMITTED is Uncommitted Read;
- SQL_TXN_READ_COMMITTED is Cursor Stability;
- SQL_TXN_REPEATABLE_READ is Read Stability;
- SQL_TXN_SERIALIZABLE is Repeatable Read.

SQL_DESCRIBE_PARAMETER (string)
″Y″ if parameters can be described; ″N″ if not.

SQL_DM_VER (string)
Reserved.

SQL_DRIVER_HDBC (32 bits)
DB2 CLI's database handle

SQL_DRIVER_HDESC (32 bits)
DB2 CLI's descriptor handle

SQL_DRIVER_HENV (32 bits)
DB2 CLI's environment handle

SQL_DRIVER_HLIB (32 bits)
Reserved.

SQL_DRIVER_HSTMT (32 bits)
DB2 CLI's statement handle

In an ODBC environment with an ODBC Driver Manager, if *InfoType* is set to SQL_DRIVER_HSTMT, the Driver Manager statement handle (i.e. the one returned from `SQLAllocStmt()`) must be passed on input in *rgbInfoValue* from the application. In this case *rgbInfoValue* is both an input and an output argument. The ODBC Driver Manager is responsible for returning the mapped value. ODBC applications wishing to call DB2 CLI specific functions (such as the LOB functions) can access them, by passing these handle values to the functions after loading the DB2 CLI library and issuing an operating system call to invoke the desired functions.

SQL_DRIVER_NAME (string)
The file name of the DB2 CLI implementation.

SQL_DRIVER_ODBC_VER (string)
The version number of ODBC that the Driver supports. DB2 CLI will return ″03.00″.

SQL_DRIVER_VER (string)
The version of the CLI driver. A string of the form 'mm.vv.rrrr' where mm is the major version, vv is the minor version and rrrr is the release. For example, ″05.01.0000″ translates to major version 5, minor version 1, release 0.

SQL_DROP_ASSERTION (32-bit unsigned integer)
Indicates which clause in the DROP ASSERTION statement is supported by the DBMS. DB2 CLI always returns zero; the DROP ASSERTION statement is not supported.

ODBC also defines the following value that is not returned by DB2 CLI:

- SQL_DA_DROP_ASSERTION

SQL_DROP_CHARACTER_SET (32-bit unsigned integer)
Indicates which clause in the DROP CHARACTER SET statement is supported by the DBMS. DB2 CLI always returns zero; the DROP CHARACTER SET statement is not supported.

ODBC also defines the following value that is not returned by DB2 CLI:

- SQL_DCS_DROP_CHARACTER_SET

SQL_DROP_COLLATION (32-bit unsigned integer)
Indicates which clause in the DROP COLLATION statement is

supported by the DBMS. DB2 CLI always returns zero; the DROP COLLATION statement is not supported.

ODBC also defines the following value that is not returned by DB2 CLI:

- SQL_DC_DROP_COLLATION

SQL_DROP_DOMAIN (32-bit unsigned integer)
Indicates which clauses in the DROP DOMAIN statement are supported by the DBMS. DB2 CLI always returns zero; the DROP DOMAIN statement is not supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_DD_DROP_DOMAIN
- SQL_DD_CASCADE
- SQL_DD_RESTRICT

SQL_DROP_SCHEMA (32-bit unsigned integer)
Indicates which clauses in the DROP SCHEMA statement are supported by the DBMS. DB2 CLI always returns zero; the DROP SCHEMA statement is not supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_DS_DROP_SCHEMA
- SQL_DS_CASCADE
- SQL_DS_RESTRICT

SQL_DROP_TABLE (32-bit unsigned integer)
Indicates which clauses in the DROP TABLE statement are supported by the DBMS:

- SQL_DT_DROP_TABLE
- SQL_DT_CASCADE
- SQL_DT_RESTRICT

SQL_DROP_TRANSLATION (32-bit unsigned integer)
Indicates which clauses in the DROP TRANSLATION statement are supported by the DBMS. DB2 CLI always returns zero; the DROP TRANSLATION statement is not supported.

ODBC also defines the following value that is not returned by DB2 CLI:

- SQL_DTR_DROP_TRANSLATION

SQL_DROP_VIEW (32-bit unsigned integer)
Indicates which clauses in the DROP VIEW statement are supported by the DBMS. DB2 CLI always returns zero; the DROP VIEW statement is not supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_DV_DROP_VIEW
- SQL_DV_CASCADE
- SQL_DV_RESTRICT

SQL_DTC_TRANSITION_COST (32-bit unsigned mask)
Used by Microsoft Transaction Server to determine whether or not the enlistment process for a connection is expensive. DB2 CLI returns:

- SQL_DTC_ENLIST_EXPENSIVE
- SQL_DTC_UNENLIST_EXPENSIVE

SQL_DYNAMIC_CURSOR_ATTRIBUTES1 (32-bit mask)
Indicates the attributes of a dynamic cursor that are supported by DB2 CLI (subset 1 of 2).

- SQL_CA1_NEXT
- SQL_CA1_ABSOLUTE
- SQL_CA1_RELATIVE
- SQL_CA1_BOOKMARK
- SQL_CA1_LOCK_EXCLUSIVE
- SQL_CA1_LOCK_NO_CHANGE
- SQL_CA1_LOCK_UNLOCK
- SQL_CA1_POS_POSITION
- SQL_CA1_POS_UPDATE
- SQL_CA1_POS_DELETE
- SQL_CA1_POS_REFRESH
- SQL_CA1_POSITIONED_UPDATE
- SQL_CA1_POSITIONED_DELETE
- SQL_CA1_SELECT_FOR_UPDATE
- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_BULK_DELETE_BY_BOOKMARK
- SQL_CA1_BULK_FETCH_BY_BOOKMARK

SQL_DYNAMIC_CURSOR_ATTRIBUTES2 (32-bit mask)
Indicates the attributes of a dynamic cursor that are supported by DB2 CLI (subset 2 of 2).

- SQL_CA2_READ_ONLY_CONCURRENCY
- SQL_CA2_LOCK_CONCURRENCY
- SQL_CA2_OPT_ROWVER_CONCURRENCY
- SQL_CA2_OPT_VALUES_CONCURRENCY
- SQL_CA2_SENSITIVITY_ADDITIONS
- SQL_CA2_SENSITIVITY_DELETIONS
- SQL_CA2_SENSITIVITY_UPDATES
- SQL_CA2_MAX_ROWS_SELECT
- SQL_CA2_MAX_ROWS_INSERT
- SQL_CA2_MAX_ROWS_DELETE
- SQL_CA2_MAX_ROWS_UPDATE
- SQL_CA2_MAX_ROWS_CATALOG
- SQL_CA2_MAX_ROWS_AFFECTS_ALL
- SQL_CA2_CRC_EXACT
- SQL_CA2_CRC_APPROXIMATE
- SQL_CA2_SIMULATE_NON_UNIQUE
- SQL_CA2_SIMULATE_TRY_UNIQUE
- SQL_CA2_SIMULATE_UNIQUE

SQL_EXPRESSIONS_IN_ORDERBY (string)
The character string ″Y″ indicates the database server supports the DIRECT specification of expressions in the ORDER BY list, ″N″ indicates that is does not.

SQL_FETCH_DIRECTION (32-bit mask)
The supported fetch directions.

The following bit-masks are used in conjunction with the flag to determine which options are supported.

- SQL_FD_FETCH_NEXT
- SQL_FD_FETCH_FIRST
- SQL_FD_FETCH_LAST
- SQL_FD_FETCH_PREV
- SQL_FD_FETCH_ABSOLUTE
- SQL_FD_FETCH_RELATIVE
- SQL_FD_FETCH_RESUME

SQL_FILE_USAGE (16-bit integer)
Indicates how a single-tier driver directly treats files in a data source. The DB2 CLI driver is not a single-tier driver and therefor always returns SQL_FILE_NOT_SUPPORTED.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_FILE_TABLE
- SQL_FILE_CATALOG

SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1 (32-bit mask)
Indicates the attributes of a forward-only cursor that are supported by DB2 CLI (subset 1 of 2).

- SQL_CA1_NEXT
- SQL_CA1_POSITIONED_UPDATE
- SQL_CA1_POSITIONED_DELETE
- SQL_CA1_SELECT_FOR_UPDATE
- SQL_CA1_LOCK_EXCLUSIVE
- SQL_CA1_LOCK_NO_CHANGE
- SQL_CA1_LOCK_UNLOCK
- SQL_CA1_POS_POSITION
- SQL_CA1_POS_UPDATE
- SQL_CA1_POS_DELETE
- SQL_CA1_POS_REFRESH
- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_BULK_DELETE_BY_BOOKMARK
- SQL_CA1_BULK_FETCH_BY_BOOKMARK

SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2 (32-bit mask)
Indicates the attributes of a forward-only cursor that are supported by DB2 CLI (subset 2 of 2).

- SQL_CA2_READ_ONLY_CONCURRENCY
- SQL_CA2_LOCK_CONCURRENCY
- SQL_CA2_MAX_ROWS_SELECT
- SQL_CA2_MAX_ROWS_CATALOG
- SQL_CA2_OPT_ROWVER_CONCURRENCY
- SQL_CA2_OPT_VALUES_CONCURRENCY
- SQL_CA2_SENSITIVITY_ADDITIONS
- SQL_CA2_SENSITIVITY_DELETIONS
- SQL_CA2_SENSITIVITY_UPDATES
- SQL_CA2_MAX_ROWS_INSERT
- SQL_CA2_MAX_ROWS_DELETE
- SQL_CA2_MAX_ROWS_UPDATE

- SQL_CA2_MAX_ROWS_AFFECTS_ALL
- SQL_CA2_CRC_EXACT
- SQL_CA2_CRC_APPROXIMATE
- SQL_CA2_SIMULATE_NON_UNIQUE
- SQL_CA2_SIMULATE_TRY_UNIQUE
- SQL_CA2_SIMULATE_UNIQUE

SQL_GETDATA_EXTENSIONS (32-bit mask)
Indicates whether extensions to the SQLGetData() function are supported. The following extensions are currently identified and supported by DB2 CLI:

- SQL_GD_ANY_COLUMN, SQLGetData() can be called for unbound columns that precede the last bound column.
- SQL_GD_ANY_ORDER, SQLGetData() can be called for columns in any order.

ODBC also defines the following extensions which are not returned by DB2 CLI:

- SQL_GD_BLOCK
- SQL_GD_BOUND

SQL_GROUP_BY (16-bit integer)
Indicates the degree of support for the GROUP BY clause by the server:

- SQL_GB_NO_RELATION, there is no relationship between the columns in the GROUP BY and in the SELECT list
- SQL_GB_NOT_SUPPORTED, GROUP BY not supported
- SQL_GB_GROUP_BY_EQUALS_SELECT, GROUP BY must include all non-aggregated columns in the select list.
- SQL_GB_GROUP_BY_CONTAINS_SELECT, the GROUP BY clause must contain all non-aggregated columns in the SELECT list.
- SQL_GB_COLLATE, a COLLATE clause can be specified at the end of each grouping column.

SQL_IDENTIFIER_CASE (16-bit integer)
Indicates case sensitivity of object names (such as table-name).

A value of:

- SQL_IC_UPPER = identifier names are stored in upper case in the system catalog.
- SQL_IC_LOWER = identifier names are stored in lower case in the system catalog.
- SQL_IC_SENSITIVE = identifier names are case sensitive, and are stored in mixed case in the system catalog.

- SQL_IC_MIXED = identifier names are not case sensitive, and are stored in mixed case in the system catalog.

Note: Identifier names in IBM DBMSs are not case sensitive.

SQL_IDENTIFIER_QUOTE_CHAR (string)
Indicates the character used to surround a delimited identifier

SQL_INDEX_KEYWORDS (32-bit mask)
Indicates the keywords in the CREATE INDEX statement that are supported:

- SQL_IK_NONE, none of the keywords are supported.
- SQL_IK_ASC, ASC keyword is supported.
- SQL_IK_DESC, DESC keyword is supported.
- SQL_IK_ALL, all keywords are supported.

To see the the CREATE INDEX statement is supported, an application can call `SQLGetInfo()` with the SQL_DLL_INDEX *InfoType*.

SQL_INFO_SCHEMA_VIEWS (32-bit mask)
Indicates the views in the INFORMATIONAL_SCHEMA that are supported. DB2 CLI always returns zero; no views in the INFORMATIONAL_SCHEMA are supported.

ODBC also defines the following values that are not returned by DB2 CLI:

- SQL_ISV_ASSERTIONS
- SQL_ISV_CHARACTER_SETS
- SQL_ISV_CHECK_CONSTRAINTS
- SQL_ISV_COLLATIONS
- SQL_ISV_COLUMN_DOMAIN_USAGE
- SQL_ISV_COLUMN_PRIVILEGES
- SQL_ISV_COLUMNS
- SQL_ISV_CONSTRAINT_COLUMN_USAGE
- SQL_ISV_CONSTRAINT_TABLE_USAGE
- SQL_ISV_DOMAIN_CONSTRAINTS
- SQL_ISV_DOMAINS
- SQL_ISV_KEY_COLUMN_USAGE
- SQL_ISV_REFERENTIAL_CONSTRAINTS
- SQL_ISV_SCHEMATA
- SQL_ISV_SQL_LANGUAGES
- SQL_ISV_TABLE_CONSTRAINTS
- SQL_ISV_TABLE_PRIVILEGES

- SQL_ISV_TABLES
- SQL_ISV_TRANSLATIONS
- SQL_ISV_USAGE_PRIVILEGES
- SQL_ISV_VIEW_COLUMN_USAGE
- SQL_ISV_VIEW_TABLE_USAGE
- SQL_ISV_VIEWS

SQL_INSERT_STATEMENT (32-bit mask)
Indicates support for INSERT statements:

- SQL_IS_INSERT_LITERALS
- SQL_IS_INSERT_SEARCHED
- SQL_IS_SELECT_INTO

SQL_INTEGRITY (string)
The ″Y″ character string indicates that the data source supports Integrity Enhanced Facility (IEF) in SQL89 and in X/Open XPG4 Embedded SQL, an ″N″ indicates it does not.

In previous versions of DB2 CLI this *InfoType* was SQL_ODBC_SQL_OPT_IEF.

SQL_KEYSET_CURSOR_ATTRIBUTES1 (32-bit mask)
Indicates the attributes of a keyset cursor that are supported by DB2 CLI (subset 1 of 2).

- SQL_CA1_NEXT
- SQL_CA1_ABSOLUTE
- SQL_CA1_RELATIVE
- SQL_CA1_BOOKMARK
- SQL_CA1_LOCK_EXCLUSIVE
- SQL_CA1_LOCK_NO_CHANGE
- SQL_CA1_LOCK_UNLOCK
- SQL_CA1_POS_POSITION
- SQL_CA1_POS_UPDATE
- SQL_CA1_POS_DELETE
- SQL_CA1_POS_REFRESH
- SQL_CA1_POSITIONED_UPDATE
- SQL_CA1_POSITIONED_DELETE
- SQL_CA1_SELECT_FOR_UPDATE
- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_BULK_DELETE_BY_BOOKMARK

- SQL_CA1_BULK_FETCH_BY_BOOKMARK

SQL_KEYSET_CURSOR_ATTRIBUTES2 (32-bit mask)
Indicates the attributes of a keyset cursor that are supported by DB2 CLI (subset 2 of 2).

- SQL_CA2_READ_ONLY_CONCURRENCY
- SQL_CA2_LOCK_CONCURRENCY
- SQL_CA2_OPT_ROWVER_CONCURRENCY
- SQL_CA2_OPT_VALUES_CONCURRENCY
- SQL_CA2_SENSITIVITY_ADDITIONS
- SQL_CA2_SENSITIVITY_DELETIONS
- SQL_CA2_SENSITIVITY_UPDATES
- SQL_CA2_MAX_ROWS_SELECT
- SQL_CA2_MAX_ROWS_INSERT
- SQL_CA2_MAX_ROWS_DELETE
- SQL_CA2_MAX_ROWS_UPDATE
- SQL_CA2_MAX_ROWS_CATALOG
- SQL_CA2_MAX_ROWS_AFFECTS_ALL
- SQL_CA2_CRC_EXACT
- SQL_CA2_CRC_APPROXIMATE
- SQL_CA2_SIMULATE_NON_UNIQUE
- SQL_CA2_SIMULATE_TRY_UNIQUE
- SQL_CA2_SIMULATE_UNIQUE

SQL_KEYWORDS (sting)
This is a string of all the *keywords* at the DBMS that are not in the ODBC's list of reserved words.

SQL_LIKE_ESCAPE_CLAUSE (string)
A character string that indicates if an escape character is supported for the metacharacters percent and underscore in a LIKE predicate.

SQL_LOCK_TYPES (32-bit mask)
Reserved option, zero is returned for the bit-mask.

SQL_MAX_ASYNC_CONCURRENT_STATEMENTS (32-bit unsigned integer)
The maximum number of active concurrent statements in asynchronous mode that DB2 CLI can support on a given connection. This value is zero if there is no specific limit, or the limit is unknown.

SQL_MAX_BINARY_LITERAL_LEN (32-bit unsigned integer)
A 32-bit unsigned integer value specifying the maximum length of a hexadecimal literal in a SQL statement.

SQL_MAX_CATALOG_NAME_LEN (16-bit integer)
The maximum length of a catalog name in the data source. This value is zero if there is no maximum length, or the length is unknown.

In previous versions of DB2 CLI this *fInfoType* was SQL_MAX_QUALIFIER_NAME_LEN.

SQL_MAX_CHAR_LITERAL_LEN (32-bit unsigned integer)
The maximum length of a character literal in an SQL statement (in bytes).

SQL_MAX_COLUMN_NAME_LEN (16-bit integer)
The maximum length of a column name (in bytes)

SQL_MAX_COLUMNS_IN_GROUP_BY (16-bit integer)
Indicates the maximum number of columns that the server supports in a GROUP BY clause. Zero if no limit.

SQL_MAX_COLUMNS_IN_INDEX (16-bit integer)
Indicates the maximum number of columns that the server supports in an index. Zero if no limit.

SQL_MAX_COLUMNS_IN_ORDER_BY (16-bit integer)
Indicates the maximum number of columns that the server supports in an ORDER BY clause. Zero if no limit.

SQL_MAX_COLUMNS_IN_SELECT (16-bit integer)
Indicates the maximum number of columns that the server supports in a select list. Zero if no limit.

SQL_MAX_COLUMNS_IN_TABLE (16-bit integer)
Indicates the maximum number of columns that the server supports in a base table. Zero if no limit.

SQL_MAX_CONCURRENT_ACTIVITIES (16-bit integer)
The maximum number of active environments that the DB2 CLI driver can support. If there is no specified limit or the limit is unknown, this value is set to zero.

In previous versions of DB2 CLI this *InfoType* was SQL_ACTIVE_ENVIRONMENTS.

SQL_MAX_CURSOR_NAME_LEN (16-bit integer)
The maximum length of a cursor name (in bytes). This value is zero if there is no maximum length, or the length is unknown.

SQL_MAX_DRIVER_CONNECTIONS (16-bit integer)
The maximum number of active connections supported per application.

Zero is returned, indicating that the limit is dependent on system resources.

The MAXCONN keyword in the db2cli.ini initialization file or the SQL_ATTR_MAX_CONNECTIONS environment/connection option can be used to impose a limit on the number of connections. This limit is returned if it is set to any value other than zero.

In previous versions of DB2 CLI this *InfoType* was SQL_ACTIVE_CONNECTIONS.

SQL_MAX_IDENTIFIER_LEN (16-bit integer)
The maximum size (in characters) that the data source supports for user-defined names.

SQL_MAX_INDEX_SIZE (32-bit unsigned integer)
Indicates the maximum size in bytes that the server supports for the combined columns in an index. Zero if no limit.

SQL_MAX_OWNER_NAME_LEN (16-bit integer)
This *fInfoType* has been replaced with SQL_MAX_SCHEMA_NAME_LEN.

The maximum length of a schema qualifier name (in bytes).

SQL_MAX_PROCEDURE_NAME_LEN (16-bit integer)
The maximum length of a procedure name (in bytes).

SQL_MAX_QUALIFIER_NAME_LEN (16-bit integer)
This *fInfoType* has been replaced with SQL_MAX_CATALOG_NAME_LEN.

The maximum length of a catalog qualifier name; first part of a 3 part table name (in bytes).

SQL_MAX_ROW_SIZE (32-bit unsigned integer)
Specifies the maximum length in bytes that the server supports in single row of a base table. Zero if no limit.

SQL_MAX_ROW_SIZE_INCLUDES_LONG (string)
Set to ″Y″ to indicate that the value returned by SQL_MAX_ROW_SIZE *InfoType* includes the length of product-specific *long string* data types. Otherwise, set to ″N″.

SQL_MAX_SCHEMA_NAME_LEN (16-bit integer)
The maximum length of a schema qualifier name (in bytes).

In previous versions of DB2 CLI this *fInfoType* was SQL_MAX_OWNER_NAME_LEN.

SQL_MAX_STATEMENT_LEN (32-bit unsigned integer)
Indicates the maximum length of an SQL statement string in bytes, including the number of white spaces in the statement.

SQL_MAX_TABLE_NAME_LEN (16-bit integer)
The maximum length of a table name (in bytes).

SQL_MAX_TABLES_IN_SELECT (16-bit integer)
Indicates the maximum number of table names allowed in a FROM clause in a <query specification>.

SQL_MAX_USER_NAME_LEN (16-bit integer)
Indicates the maximum size allowed for a <user identifier> (in bytes).

SQL_MULT_RESULT_SETS (string)
The character string ″Y″ indicates that the database supports multiple result sets, ″N″ indicates that it does not.

SQL_MULTIPLE_ACTIVE_TXN (string)
The character string ″Y″ indicates that active transactions on multiple connections are allowed, ″N″ indicates that only one connection at a time can have an active transaction.

DB2 CLI returns ″N″ for coordinated distributed unit of work (CONNECT TYPE 2) connections, (since the transaction or Unit Of Work spans all connections), and returns ″Y″ for all other connections.

SQL_NEED_LONG_DATA_LEN (string)
A character string reserved for the use of ODBC. “N” is always returned.

SQL_NON_NULLABLE_COLUMNS (16-bit integer)
Indicates whether non-nullable columns are supported:

- SQL_NNC_NON_NULL, columns can be defined as NOT NULL.
- SQL_NNC_NULL, columns can not be defined as NOT NULL.

SQL_NULL_COLLATION (16-bit integer)
Indicates where NULLs are sorted in a list:

- SQL_NC_HIGH, null values sort high
- SQL_NC_LOW, to indicate that null values sort low

SQL_NUMERIC_FUNCTIONS (32-bit mask)
Indicates the ODBC scalar numeric functions supported These functions are intended to be used with the ODBC vendor escape sequence described in “Using Vendor Escape Clauses” on page 144.

The following bit-masks are used to determine which numeric functions are supported:

- SQL_FN_NUM_ABS
- SQL_FN_NUM_ACOS
- SQL_FN_NUM_ASIN
- SQL_FN_NUM_ATAN
- SQL_FN_NUM_ATAN2
- SQL_FN_NUM_CEILING

- SQL_FN_NUM_COS
- SQL_FN_NUM_COT
- SQL_FN_NUM_DEGREES
- SQL_FN_NUM_EXP
- SQL_FN_NUM_FLOOR
- SQL_FN_NUM_LOG
- SQL_FN_NUM_LOG10
- SQL_FN_NUM_MOD
- SQL_FN_NUM_PI
- SQL_FN_NUM_POWER
- SQL_FN_NUM_RADIANS
- SQL_FN_NUM_RAND
- SQL_FN_NUM_ROUND
- SQL_FN_NUM_SIGN
- SQL_FN_NUM_SIN
- SQL_FN_NUM_SQRT
- SQL_FN_NUM_TAN
- SQL_FN_NUM_TRUNCATE

SQL_ODBC_API_CONFORMANCE (16-bit integer)
The level of ODBC conformance.

- SQL_OAC_NONE
- SQL_OAC_LEVEL1
- SQL_OAC_LEVEL2

SQL_ODBC_INTERFACE_CONFORMANCE (32-bit unsigned integer)
Indicates the level of the ODBC 3.0 interface that the DB2 CLI driver conforms to:

- SQL_OIC_CORE, the minimum level that all ODBC drivers are expected to conform to. This level includes basic interface elements such as connection functions; functions for preparing and executing an SQL statement; basic result set metadata functions; basic catalog functions; and so on.
- SQL_OIC_LEVEL1, a level including the core standards compliance level functionality, plus scrollable cursors, bookmarks, positioned updates and deletes, and so on.
- SQL_OIC_LEVEL2, a level including level 1 standards compliance level functionality, plus advanced features such as sensitive cursors; update, delete, and refresh by bookmarks; stored procedure support; catalog functions for primary and foreign keys; multi-catalog support; and so on.

SQL_SCHEMA_TERM (string)
The database vendor's terminology for a schema (owner).

In previous versions of DB2 CLI this *InfoType* was SQL_OWNER_TERM.

SQL_SCHEMA_USAGE (32-bit mask)
Indicates the type of SQL statements that have schema (owners) associated with them when these statements are executed, Schema qualifiers (owners) are:

- SQL_SU_DML_STATEMENTS - supported in all DML statements.
- SQL_SU_PROCEDURE_INVOCATION - supported in the procedure invocation statement.
- SQL_SU_TABLE_DEFINITION - supported in all table definition statements.
- SQL_SU_INDEX_DEFINITION - supported in all index definition statements.
- SQL_SU_PRIVILEGE_DEFINITION - supported in all privilege definition statements (i.e. grant and revoke statements).

In previous versions of DB2 CLI this *InfoType* was SQL_OWNER_USAGE.

SQL_ODBC_SAG_CLI_CONFORMANCE (16-bit integer)
The compliance to the functions of the SQL Access Group (SAG) CLI specification.

A value of:

- SQL_OSCC_NOT_COMPLIANT - the driver is not SAG-compliant.
- SQL_OSCC_COMPLIANT - the driver is SAG-compliant.

SQL_ODBC_SQL_CONFORMANCE (16-bit integer)
A value of:

- SQL_OSC_MINIMUM, minimum ODBC SQL grammar supported
- SQL_OSC_CORE, core ODBC SQL Grammar supported
- SQL_OSC_EXTENDED, extended ODBC SQL Grammar supported

For the definition of the above 3 types of ODBC SQL grammar, see the *ODBC 3.0 Software Development Kit and Programmer's Reference*

SQL_ODBC_SQL_OPT_IEF (string)
This *InfoType* has been replaced with SQL_INTEGRITY.

The ″Y″ character string indicates that the data source supports Integrity Enhanced Facility (IEF) in SQL89 and in X/Open XPG4 Embedded SQL, an ″N″ indicates it does not.

SQL_ODBC_VER (string)
The version number of ODBC that the driver manager supports.

DB2 CLI will return the string ″03.01.0000″.

SQL_OJ_CAPABILITIES (32-bit mask)
A 32-bit bit-mask enumerating the types of outer join supported.

The bitmasks are:

- SQL_OJ_LEFT : Left outer join is supported.
- SQL_OJ_RIGHT : Right outer join is supported.
- SQL_OJ_FULL : Full outer join is supported.
- SQL_OJ_NESTED : Nested outer join is supported.
- SQL_OJ_ORDERED : The order of the tables underlying the columns in the outer join ON clause need not be in the same order as the tables in the JOIN clause.
- SQL_OJ_INNER : The inner table of an outer join can also be an inner join.
- SQL_OJ_ALL_COMPARISONS : Any predicate may be used in the outer join ON clause. If this bit is not set, the equality (=) operator is the only valid comparison operator in the ON clause.

SQL_ORDER_BY_COLUMNS_IN_SELECT (string)
Set to ″Y″ if columns in the ORDER BY clauses must be in the select list; otherwise set to ″N″.

SQL_OUTER_JOINS (string)
The character string:

- ″Y″ indicates that outer joins are supported, and DB2 CLI supports the ODBC outer join request syntax.
- ″N″ indicates that it is not supported.

(See “Using Vendor Escape Clauses” on page 144)

SQL_OWNER_TERM (string)
This *InfoType* has been replaced with SQL_SCHEMA_TERM.

The database vendor’s terminology for a schema (owner).

SQL_OWNER_USAGE (32-bit mask)
This *InfoType* has been replaced with SQL_SCHEMA_USAGE.

Indicates the type of SQL statements that have schema (owners) associated with them when these statements are executed, Schema qualifiers (owners) are:

- SQL_OU_DML_STATEMENTS - supported in all DML statements.

- SQL_OU_PROCEDURE_INVOCATION - supported in the procedure invocation statement.
- SQL_OU_TABLE_DEFINITION - supported in all table definition statements.
- SQL_OU_INDEX_DEFINITION - supported in all index definition statements.
- SQL_OU_PRIVILEGE_DEFINITION - supported in all privilege definition statements (i.e. grant and revoke statements).

SQL_PARAM_ARRAY_ROW_COUNTS (32-bit unsigned integer)
Indicates the availability of row counts in a parameterized execution:

- SQL_PARC_BATCH, individual row counts are available for each set of parameters. This is conceptually equivalent to the driver generating a batch of SQL statements, one for each parameter set in the array. Extended error information can be retrieved by using the SQL_PARAM_STATUS_PTR descriptor field.
- SQL_PARC_NO_BATCH, there is only one row count available, which is the cumulative row count resulting from the execution of the statement for the entire array of parameters. This is conceptually equivalent to treating the statement along with the entire parameter array as one atomic unit. Errors are handled the same as if one statement were executed.

SQL_PARAM_ARRAY_SELECTS (32-bit unsigned integer)
Indicates the availability of result sets in a parameterized execution:

- SQL_PAS_BATCH, there is one result set available per set of parameters. This is conceptually equivalent to the driver generating a batch of SQL statements, one for each parameter set in the array.
- SQL_PAS_NO_BATCH, there is only one result set available, which represents the cumulative result set resulting from the execution of the statement for the entire array of parameters. This is conceptually equivalent to treating the statement along with the entire parameter array as one atomic unit.
- SQL_PAS_NO_SELECT, a driver does not allow a result-set generating statement to be executed with an array of parameters.

SQL_POS_OPERATIONS (32-bit mask)
Reserved option, zero is returned for the bit-mask.

SQL_POSITIONED_STATEMENTS (32-bit mask)
Indicates the degree of support for Positioned UPDATE and Positioned DELETE statements:

- SQL_PS_POSITIONED_DELETE
- SQL_PS_POSITIONED_UPDATE

- SQL_PS_SELECT_FOR_UPDATE, indicates whether or not the server requires the FOR UPDATE clause to be specified on a <query expression> in order for a column to be updateable via the cursor.

SQL_PROCEDURE_TERM (string)
The name a database vendor uses for a procedure

SQL_PROCEDURES (string)
A character string of ″Y″ indicates that the data source supports procedures and DB2 CLI supports the ODBC procedure invocation syntax specified in “Using Stored Procedures” on page 125. ″N″ indicates that it does not.

SQL_QUALIFIER_LOCATION (16-bit integer)
This *InfoType* has been replaced with SQL_CATALOG_LOCATION.

A 16-bit integer value indicated the position of the qualifier in a qualified table name. DB2 CLI always returns SQL_QL_START for this information type.

SQL_QUALIFIER_NAME_SEPARATOR (string)
The character(s) used as a separator between a catalog name and the qualified name element that follows it.

This *InfoType* has been replaced with SQL_CATALOG_NAME_SEPARATOR.

SQL_QUALIFIER_TERM (string)
The database vendor’s terminology for a qualifier

The name that the vendor uses for the high order part of a three part name.

Since DB2 CLI does not support three part names, a zero-length string is returned.

This *InfoType* has been replaced with SQL_CATALOG_TERM.

SQL_QUALIFIER_USAGE (32-bit mask)
This *fInfoType* has been replaced with SQL_CATALOG_USAGE.

This is similar to SQL_OWNER_USAGE except that this is used for catalog.

SQL_QUOTED_IDENTIFIER_CASE (16-bit integer)
Returns:

- SQL_IC_UPPER - quoted identifiers in SQL are case insensitive and stored in upper case in the system catalog.
- SQL_IC_LOWER - quoted identifiers in SQL are case insensitive and are stored in lower case in the system catalog.

- SQL_IC_SENSITIVE - quoted identifiers (delimited identifiers) in SQL are case sensitive and are stored in mixed case in the system catalog.
- SQL_IC_MIXED - quoted identifiers in SQL are case insensitive and are stored in mixed case in the system catalog.

This should be contrasted with the SQL_IDENTIFIER_CASE *InfoType* which is used to determine how (unquoted) identifiers are stored in the system catalog.

SQL_ROW_UPDATES (string)
A character string of ″Y″ indicates changes are detected in rows between multiple fetches of the same rows, ″N″ indicates that changes are not detected.

SQL_SCROLL_CONCURRENCY (32-bit mask)
Indicates the concurrency options supported for the cursor.

The following bit-masks are used in conjunction with the flag to determine which options are supported:

- SQL_SCCO_READ_ONLY
- SQL_SCCO_LOCK
- SQL_SCCO_TIMESTAMP
- SQL_SCCO_VALUES

DB2 CLI returns SQL_SCCO_LOCK. indicating that the lowest level of locking that is sufficient to ensure the row can be updated is used.

SQL_SCROLL_OPTIONS (32-bit mask)
The scroll options supported for scrollable cursors.

The following bit-masks are used in conjunction with the flag to determine which options are supported:

- SQL_SO_FORWARD_ONLY
- SQL_SO_KEYSET_DRIVEN
- SQL_SO_STATIC
- SQL_SO_DYNAMIC
- SQL_SO_MIXED

For more information about scrollable cursors see “Scrollable Cursors” on page 68.

SQL_SEARCH_PATTERN_ESCAPE (string)
Used to specify what the driver supports as an escape character for catalog functions such as (`SQLTables()`, `SQLColumns()`)

SQL_SERVER_NAME (string)
The Name of the DB2 Instance. In contrast to SQL_DATA_SOURCE_NAME, this is the actual name of the database server. (Some DBMSs provide a different name on CONNECT than the real server-name of the database.)

SQL_SPECIAL_CHARACTERS (string)
Contains all the characters in addition to `a...z`, `A...Z`, `0...9`, and `_` that the server allows in non-delimited identifiers.

SQL_SQL_CONFORMANCE (32-bit unsigned integer)
Indicates the level of SQL-92 supported:

- SQL_SC_SQL92_ENTRY, entry level SQL-92 compliant.
- SQL_SC_FIPS127_2_TRANSITIONAL, FIPS 127-2 transitional level compliant.
- SQL_SC_SQL92_FULL, full level SQL-92 compliant.
- SQL_SC_ SQL92_INTERMEDIATE, intermediate level SQL-92 compliant.

SQL_SQL92_DATETIME_FUNCTIONS (32-bit mask)
Indicates the datetime scalar functions that are supported by DB2 CLI and the data source:

- SQL_SDF_CURRENT_DATE
- SQL_SDF_CURRENT_TIME
- SQL_SDF_CURRENT_TIMESTAMP

SQL_SQL92_FOREIGN_KEY_DELETE_RULE (32-bit mask)
Indicates the rules supported for a foreign key in a DELETE statement, as defined by SQL-92:

- SQL_SFKD_CASCADE
- SQL_SFKD_NO_ACTION
- SQL_SFKD_SET_DEFAULT
- SQL_SFKD_SET_NULL

SQL_SQL92_FOREIGN_KEY_UPDATE_RULE (32-bit mask)
Indicates the rules supported for a foreign key in an UPDATE statement, as defined by SQL-92:

- SQL_SFKU_CASCADE
- SQL_SFKU_NO_ACTION
- SQL_SFKU_SET_DEFAULT
- SQL_SFKU_SET_NULL

SQL_SQL92_GRANT (32-bit mask)
Indicates the clauses supported in a GRANT statement, as defined by SQL-92:

- SQL_SG_DELETE_TABLE
- SQL_SG_INSERT_COLUMN
- SQL_SG_INSERT_TABLE
- SQL_SG_REFERENCES_TABLE
- SQL_SG_REFERENCES_COLUMN
- SQL_SG_SELECT_TABLE
- SQL_SG_UPDATE_COLUMN
- SQL_SG_UPDATE_TABLE
- SQL_SG_USAGE_ON_DOMAIN
- SQL_SG_USAGE_ON_CHARACTER_SET
- SQL_SG_USAGE_ON_COLLATION
- SQL_SG_USAGE_ON_TRANSLATION
- SQL_SG_WITH_GRANT_OPTION

SQL_SQL92_NUMERIC_VALUE_FUNCTIONS (32-bit mask)
Indicates the numeric value scalar functions that are supported by DB2 CLI and the data source, as defined in SQL-92:

- SQL_SNVF_BIT_LENGTH
- SQL_SNVF_CHAR_LENGTH
- SQL_SNVF_CHARACTER_LENGTH
- SQL_SNVF_EXTRACT
- SQL_SNVF_OCTET_LENGTH
- SQL_SNVF_POSITION

SQL_SQL92_PREDICATES (32-bit mask)
Indicates the predicates supported in a SELECT statement, as defined by SQL-92.

- SQL_SP_BETWEEN
- SQL_SP_COMPARISON
- SQL_SP_EXISTS
- SQL_SP_IN
- SQL_SP_ISNOTNULL
- SQL_SP_ISNULL
- SQL_SP_LIKE
- SQL_SP_MATCH_FULL
- SQL_SP_MATCH_PARTIAL
- SQL_SP_MATCH_UNIQUE_FULL
- SQL_SP_MATCH_UNIQUE_PARTIAL
- SQL_SP_OVERLAPS

- SQL_SP_QUANTIFIED_COMPARISON
- SQL_SP_UNIQUE

SQL_SQL92_RELATIONAL_JOIN_OPERATORS (32-bit mask)
Indicates the relational join operators supported in a SELECT statement, as defined by SQL-92.

- SQL_SRJO_CORRESPONDING_CLAUSE
- SQL_SRJO_CROSS_JOIN
- SQL_SRJO_EXCEPT_JOIN
- SQL_SRJO_FULL_OUTER_JOIN
- SQL_SRJO_INNER_JOIN (indicates support for the INNER JOIN syntax, not for the inner join capability)
- SQL_SRJO_INTERSECT_JOIN
- SQL_SRJO_LEFT_OUTER_JOIN
- SQL_SRJO_NATURAL_JOIN
- SQL_SRJO_RIGHT_OUTER_JOIN
- SQL_SRJO_UNION_JOIN

SQL_SQL92_REVOKE (32-bit mask)
Indicates which clauses the data source supports in the REVOKE statement, as defined by SQL-92:

- SQL_SR_CASCADE
- SQL_SR_DELETE_TABLE
- SQL_SR_GRANT_OPTION_FOR
- SQL_SR_INSERT_COLUMN
- SQL_SR_INSERT_TABLE
- SQL_SR_REFERENCES_COLUMN
- SQL_SR_REFERENCES_TABLE
- SQL_SR_RESTRICT
- SQL_SR_SELECT_TABLE
- SQL_SR_UPDATE_COLUMN
- SQL_SR_UPDATE_TABLE
- SQL_SR_USAGE_ON_DOMAIN
- SQL_SR_USAGE_ON_CHARACTER_SET
- SQL_SR_USAGE_ON_COLLATION
- SQL_SR_USAGE_ON_TRANSLATION

SQL_SQL92_ROW_VALUE_CONSTRUCTOR (32-bit mask)
Indicates the row value constructor expressions supported in a SELECT statement, as defined by SQL-92.

- SQL_SRVC_VALUE_EXPRESSION

- SQL_SRVC_NULL
- SQL_SRVC_DEFAULT
- SQL_SRVC_ROW_SUBQUERY

SQL_SQL92_STRING_FUNCTIONS (32-bit mask)
Indicates the string scalar functions that are supported by DB2 CLI and the data source, as defined by SQL-92:

- SQL_SSF_CONVERT
- SQL_SSF_LOWER
- SQL_SSF_UPPER
- SQL_SSF_SUBSTRING
- SQL_SSF_TRANSLATE
- SQL_SSF_TRIM_BOTH
- SQL_SSF_TRIM_LEADING
- SQL_SSF_TRIM_TRAILING

SQL_SQL92_VALUE_EXPRESSIONS (32-bit mask)
Indicates the value expressions supported, as defined by SQL-92.

- SQL_SVE_CASE
- SQL_SVE_CAST
- SQL_SVE_COALESCE
- SQL_SVE_NULLIF

SQL_SQL92_STANDARD_CLI_CONFORMANCE (32-bit mask)
Indicates the CLI standard or standards to which DB2 CLI conforms:

- SQL_SCC_XOPEN_CLI_VERSION1
- SQL_SCC_ISO92_CLI

SQL_STATIC_CURSOR_ATTRIBUTES1 (32-bit mask)
Indicates the attributes of a static cursor that are supported by DB2 CLI (subset 1 of 2):

- SQL_CA1_NEXT
- SQL_CA1_ABSOLUTE
- SQL_CA1_RELATIVE
- SQL_CA1_BOOKMARK
- SQL_CA1_LOCK_NO_CHANGE
- SQL_CA1_LOCK_EXCLUSIVE
- SQL_CA1_LOCK_UNLOCK
- SQL_CA1_POS_POSITION
- SQL_CA1_POS_UPDATE
- SQL_CA1_POS_DELETE

- SQL_CA1_POS_REFRESH
- SQL_CA1_POSITIONED_UPDATE
- SQL_CA1_POSITIONED_DELETE
- SQL_CA1_SELECT_FOR_UPDATE
- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_BULK_DELETE_BY_BOOKMARK
- SQL_CA1_BULK_FETCH_BY_BOOKMARK

SQL_STATIC_CURSOR_ATTRIBUTES2 (32-bit mask)
Indicates the attributes of a static cursor that are supported by DB2 CLI (subset 2 of 2):

- SQL_CA2_READ_ONLY_CONCURRENCY
- SQL_CA2_LOCK_CONCURRENCY
- SQL_CA2_OPT_ROWVER_CONCURRENCY
- SQL_CA2_OPT_VALUES_CONCURRENCY
- SQL_CA2_SENSITIVITY_ADDITIONS
- SQL_CA2_SENSITIVITY_DELETIONS
- SQL_CA2_SENSITIVITY_UPDATES
- SQL_CA2_MAX_ROWS_SELECT
- SQL_CA2_MAX_ROWS_INSERT
- SQL_CA2_MAX_ROWS_DELETE
- SQL_CA2_MAX_ROWS_UPDATE
- SQL_CA2_MAX_ROWS_CATALOG
- SQL_CA2_MAX_ROWS_AFFECTS_ALL
- SQL_CA2_CRC_EXACT
- SQL_CA2_CRC_APPROXIMATE
- SQL_CA2_SIMULATE_NON_UNIQUE
- SQL_CA2_SIMULATE_TRY_UNIQUE
- SQL_CA2_SIMULATE_UNIQUE

SQL_STATIC_SENSITIVITY (32-bit mask)
Indicates whether changes made by an application with a positioned update or delete statement can be detected by that application:

- SQL_SS_ADDITIONS: Added rows are visible to the cursor; the cursor can scroll to these rows. All DB2 servers see added rows.
- SQL_SS_DELETIONS: Deleted rows are no longer available to the cursor and do not leave a hole in the result set; after the cursor scrolls from a deleted row, it cannot return to that row.

- SQL_SS_UPDATES: Updates to rows are visible to the cursor; if the cursor scrolls from and returns to an updated row, the data returned by the cursor is the updated data, not the original data.

SQL_STRING_FUNCTIONS (32-bit mask)

Indicates which string functions are supported.

The following bit-masks are used to determine which string functions are supported:

- SQL_FN_STR_ASCII
- SQL_FN_STR_BIT_LENGTH
- SQL_FN_STR_CHAR
- SQL_FN_STR_CHAR_LENGTH
- SQL_FN_STR_CHARACTER_LENGTH
- SQL_FN_STR_CONCAT
- SQL_FN_STR_DIFFERENCE
- SQL_FN_STR_INSERT
- SQL_FN_STR_LCASE
- SQL_FN_STR_LEFT
- SQL_FN_STR_LENGTH
- SQL_FN_STR_LOCATE
- SQL_FN_STR_LOCATE_2
- SQL_FN_STR_LTRIM
- SQL_FN_STR_OCTET_LENGTH
- SQL_FN_STR_POSITION
- SQL_FN_STR_REPEAT
- SQL_FN_STR_REPLACE
- SQL_FN_STR_RIGHT
- SQL_FN_STR_RTRIM
- SQL_FN_STR_SOUNDEX
- SQL_FN_STR_SPACE
- SQL_FN_STR_SUBSTRING
- SQL_FN_STR_UCASE

If an application can call the LOCATE scalar function with the *string_exp1*, *string_exp2*, and *start* arguments, the SQL_FN_STR_LOCATE bitmask is returned. If an application can only call the LOCATE scalar function with the *string_exp1* and *string_exp2*, the SQL_FN_STR_LOCATE_2 bitmask is returned. If the LOCATE scalar function is fully supported, both bitmasks are returned.

SQL_SUBQUERIES (32-bit mask)
Indicates which predicates support subqueries:

- SQL_SQ_COMPARISION - the *comparison* predicate
- SQL_SQ_CORRELATE_SUBQUERIES - all predicates
- SQL_SQ_EXISTS - the *exists* predicate
- SQL_SQ_IN - the *in* predicate
- SQL_SQ_QUANTIFIED - the predicates containing a quantification scalar function.

SQL_SYSTEM_FUNCTIONS (32-bit mask)
Indicates which scalar system functions are supported.

The following bit-masks are used to determine which scalar system functions are supported:

- SQL_FN_SYS_DBNAME
- SQL_FN_SYS_IFNULL
- SQL_FN_SYS_USERNAME

Note: These functions are intended to be used with the escape sequence in ODBC.

SQL_TABLE_TERM (string)
The database vendor's terminology for a table

SQL_TIMEDATE_ADD_INTERVALS (32-bit mask)
Indicates whether or not the special ODBC system function TIMESTAMPADD is supported, and, if it is, which intervals are supported.

The following bitmasks are used to determine which intervals are supported:

- SQL_FN_TSI_FRAC_SECOND
- SQL_FN_TSI_SECOND
- SQL_FN_TSI_MINUTE
- SQL_FN_TSI_HOUR
- SQL_FN_TSI_DAY
- SQL_FN_TSI_WEEK
- SQL_FN_TSI_MONTH
- SQL_FN_TSI_QUARTER
- SQL_FN_TSI_YEAR

SQL_TIMEDATE_DIFF_INTERVALS (32-bit mask)
Indicates whether or not the special ODBC system function TIMESTAMPDIFF is supported, and, if it is, which intervals are supported.

The following bitmasks are used to determine which intervals are supported:

- SQL_FN_TSI_FRAC_SECOND
- SQL_FN_TSI_SECOND
- SQL_FN_TSI_MINUTE
- SQL_FN_TSI_HOUR
- SQL_FN_TSI_DAY
- SQL_FN_TSI_WEEK
- SQL_FN_TSI_MONTH
- SQL_FN_TSI_QUARTER
- SQL_FN_TSI_YEAR

SQL_TIMEDATE_FUNCTIONS (32-bit mask)

Indicates which time and date functions are supported.

The following bit-masks are used to determine which date functions are supported:

- SQL_FN_TD_CURRENT_DATE
- SQL_FN_TD_CURRENT_TIME
- SQL_FN_TD_CURRENT_TIMESTAMP
- SQL_FN_TD_CURDATE
- SQL_FN_TD_CURTIME
- SQL_FN_TD_DAYNAME
- SQL_FN_TD_DAYOFMONTH
- SQL_FN_TD_DAYOFWEEK
- SQL_FN_TD_DAYOFYEAR
- SQL_FN_TD_EXTRACT
- SQL_FN_TD_HOUR
- SQL_FN_TD_JULIAN_DAY
- SQL_FN_TD_MINUTE
- SQL_FN_TD_MONTH
- SQL_FN_TD_MONTHNAME
- SQL_FN_TD_NOW
- SQL_FN_TD_QUARTER
- SQL_FN_TD_SECOND
- SQL_FN_TD_SECONDS_SINCE_MIDNIGHT
- SQL_FN_TD_TIMESTAMPADD
- SQL_FN_TD_TIMESTAMPDIFF
- SQL_FN_TD_WEEK

- SQL_FN_TD_YEAR

Note: These functions are intended to be used with the escape sequence in ODBC.

SQL_TXN_CAPABLE (16-bit integer)

Indicates whether transactions can contain DDL or DML or both.

- SQL_TC_NONE = transactions not supported.
- SQL_TC_DML = transactions can only contain DML statements (SELECT, INSERT, UPDATE, DELETE, etc.) DDL statements (CREATE TABLE, DROP INDEX, etc.) encountered in a transaction cause an error.
- SQL_TC_DDL_COMMIT = transactions can only contain DML statements. DDL statements encountered in a transaction cause the transaction to be committed.
- SQL_TC_DDL_IGNORE = transactions can only contain DML statements. DDL statements encountered in a transaction are ignored.
- SQL_TC_ALL = transactions can contain DDL and DML statements in any order.

SQL_TXN_ISOLATION_OPTION (32-bit mask)

The transaction isolation levels available at the currently connected database server.

The following masks are used in conjunction with the flag to determine which options are supported:

- SQL_TXN_READ_UNCOMMITTED
- SQL_TXN_READ_COMMITTED
- SQL_TXN_REPEATABLE_READ
- SQL_TXN_SERIALIZABLE
- SQL_TXN_NOCOMMIT
- SQL_TXN_VERSIONING

For descriptions of each level refer to SQL_DEFAULT_TXN_ISOLATION.

SQL_UNION (32-bit mask)

Indicates if the server supports the UNION operator:

- SQL_U_UNION - supports the UNION clause
- SQL_U_UNION_ALL - supports the ALL keyword in the UNION clause

If SQL_U_UNION_ALL is set, so is SQL_U_UNION.

SQL_USER_NAME (string)
The user name used in a particular database. This is the identifier specified on the SQLConnect() call.

SQL_XOPEN_CLI_YEAR (string)
Indicates the year of publication of the X/Open specification with which the version of the driver fully complies.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 112. SQLGetInfo SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The requested information was returned as a string and its length exceeded the length of the application buffer as specified in *BufferLength*. The argument *StringLengthPtr* contains the actual (not truncated) length of the requested information. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The type of information requested in *InfoType* requires an open connection. Only SQL_ODBC_VER does not require an open connection.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument *BufferLength* was less than 0.
HY096	Information type out of range.	An invalid *InfoType* was specified.
HYC00	Driver not capable.	The value specified in the argument *InfoType* is not supported by either DB2 CLI or the data source.

Restrictions

None.

Example

```
/* From CLI sample getinfo.c */
/* ... */
*/
    /* Check to see if SQLGetInfo() is supported */
    rc = SQLGetFunctions(hdbc, SQL_API_SQLGETINFO, &supported);

    if (supported == SQL_TRUE) { /* get information about current connection */

        rc = SQLGetInfo(hdbc, SQL_DATA_SOURCE_NAME, buffer, 255, &outlen);
        printf("      Server Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DATABASE_NAME,    buffer, 255, &outlen);
        printf("    Database Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_SERVER_NAME,      buffer, 255, &outlen);
        printf("    Instance Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DBMS_NAME,        buffer, 255, &outlen);
        printf("        DBMS Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DBMS_VER,         buffer, 255, &outlen);
        printf("     DBMS Version: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DRIVER_NAME,      buffer, 255, &outlen);
        printf("   CLI Driver Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DRIVER_VER,       buffer, 255, &outlen);
        printf("CLI Driver Version: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_ODBC_SQL_CONFORMANCE, &output,
                        sizeof(output), &outlen);
        switch (output) {
        case 0:
            strcpy((char *)buffer, "Minimum Grammar");
            break;
        case 1:
            strcpy((char *)buffer, "Core Grammar");
            break;
        case 2:
            strcpy((char *)buffer, "Extended Grammar");
            break;
        default:
            printf("Error calling getinfo!");
            return (SQL_ERROR);
        }
        printf("ODBC SQL Conformance Level: %s\n", buffer);
    }
    else printf( "SQLGetInfo is not supported!\n" ) ;
```

References

• “SQLGetTypeInfo - Get Data Type Information” on page 555

SQLGetLength - Retrieve Length of A String Value

Purpose

Specification:	DB2 CLI 2.1		

SQLGetLength() is used to retrieve the length of a large object value, referenced by a large object locator that has been returned from the server (as a result of a fetch, or an SQLGetSubString() call) during the current transaction.

Syntax

```
SQLRETURN  SQLGetLength      (SQLHSTMT          StatementHandle,  /* hstmt */
                              SQLSMALLINT       LocatorCType,
                              SQLINTEGER        Locator,
                              SQLINTEGER   FAR  *StringLength,
                              SQLINTEGER   FAR  *IndicatorValue);
```

Function Arguments

Table 113. SQLGetLength Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle. This can be any statement handle which has been allocated but which does not currently have a prepared statement assigned to it.
SQLSMALLINT	LocatorCType	input	The C type of the source LOB locator. This may be: • SQL_C_BLOB_LOCATOR • SQL_C_CLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR
SQLINTEGER	Locator	input	Must be set to the LOB locator value.
SQLINTEGER *	StringLength	output	The length of the returned information in *rgbValue* in bytes[a] if the target C buffer type is intended for a binary or character string variable and not a locator value. If the pointer is set to NULL then the SQLSTATE **HY**009 is returned.
SQLINTEGER *	IndicatorValue	output	Always set to zero.

Note:

a This is in bytes even for DBCLOB data.

Usage

SQLGetLength() can be used to determine the length of the data value represented by a LOB locator. It is used by applications to determine the overall length of the referenced LOB value so that the appropriate strategy to obtain some or all of the LOB value can be chosen.

The Locator argument can contain any valid LOB locator which has not been explicitly freed using a FREE LOCATOR statement nor implicitly freed because the transaction during which it was created has terminated.

The statement handle must not have been associated with any prepared statements or catalog function calls.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 114. SQLGetLength SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The combination of *LocatorCType* and *Locator* is not valid.
40003 **08**S01	Communication link failure.	
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	*LocatorCType* is not one of SQL_C_CLOB_LOCATOR, SQL_C_BLOB_LOCATOR, or SQL_C_DBCLOB_LOCATOR.
HY009	Invalid argument value.	Pointer to *StringLength* was NULL.

Table 114. SQLGetLength SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The specified *StatementHandle* is not in an *allocated* state. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.
0F001	The LOB token variable does not currently represent any value.	The value specified for *Locator* has not been associated with a LOB locator.

Restrictions

This function is not available when connected to a DB2 server that does not support Large Objects. Call `SQLGetFunctions()` with the function type set to SQL_API_SQLGETLENGTH and check the *fExists* output argument to determine if the function is supported for the current connection.

Example

Refer to “Example” on page 538.

References

- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408
- “SQLFetch - Fetch Next Row” on page 396
- “SQLGetPosition - Return Starting Position of String” on page 535
- “SQLGetSubString - Retrieve Portion of A String Value” on page 550

SQLGetPosition - Return Starting Position of String

Purpose

Specification:	DB2 CLI 2.1		

SQLGetPosition() is used to return the starting position of one string within a LOB value (the source). The source value must be a LOB locator, the search string can be a LOB locator or a literal string.

The source and search LOB locators can be any that have been returned from the database from a fetch or a SQLGetSubString() call during the current transaction.

Syntax

```
SQLRETURN   SQLGetPosition   (SQLHSTMT          StatementHandle,   /* hstmt */
                              SQLSMALLINT       LocatorCType,
                              SQLINTEGER        SourceLocator,
                              SQLINTEGER        SearchLocator,
                              SQLCHAR      FAR  *SearchLiteral,
                              SQLINTEGER        SearchLiteralLength,
                              SQLUINTEGER       FromPosition,
                              SQLUINTEGER  FAR  *LocatedAt,
                              SQLINTEGER   FAR  *IndicatorValue);
```

Function Arguments

Table 115. SQLGetPosition Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle. This can be any statement handle which has been allocated but which does not currently have a prepared statement assigned to it.
SQLSMALLINT	LocatorCType	input	The C type of the source LOB locator. This may be: • SQL_C_BLOB_LOCATOR • SQL_C_CLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR
SQLINTEGER	Locator	input	*Locator* must be set to the source LOB locator.
SQLINTEGER	SearchLocator	input	If the *SearchLiteral* pointer is NULL and if *SearchLiteralLength* is set to 0, then *SearchLocator* must be set to the LOB locator associated with the search string; otherwise, this argument is ignored.

Table 115. SQLGetPosition Arguments (continued)

Data Type	Argument	Use	Description
SQLCHAR *	SearchLiteral	input	This argument points to the area of storage that contains the search string literal. If *SearchLiteralLength* is 0, this pointer must be NULL.
SQLINTEGER	SearchLiteralLength	input	The length of the string in *SearchLiteral*(in bytes). [a] If this argument value is 0, then the argument *SearchLocator* is meaningful.
SQLUINTEGER	FromPosition	input	For BLOBs and CLOBs, this is the position of the first byte within the source string at which the search is to start. to be returned by the function. For DBCLOBs, this is the first character. The start byte or character is numbered 1.
SQLUINTEGER *	LocatedAt	output	For BLOBs and CLOBs, this is the byte position at which the string was located or, if not located, the value zero. For DBCLOBs, this is the character position. If the length of the source string is zero, the value 1 is returned.
SQLINTEGER *	IndicatorValue	output	Always set to zero.

Note:

a This is in bytes even for DBCLOB data.

Usage

`SQLGetPosition()` is used in conjunction with `SQLGetSubString()` in order to obtain any portion of a string in a random manner. In order to use `SQLGetSubString()`, the location of the substring within the overall string must be known in advance. In situations where the start of that substring can be found by a search string, `SQLGetPosition()` can be used to obtain the starting position of that substring.

The *Locator* and *SearchLocator* (if used) arguments can contain any valid LOB locator which has not been explicitly freed using a FREE LOCATOR statement or implicitly freed because the transaction during which it was created has terminated.

The *Locator* and *SearchLocator* must have the same LOB locator type.

The statement handle must not have been associated with any prepared statements or catalog function calls.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 116. SQLGetPosition SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The combination of *LocatorCType* and either of the LOB locator values is not valid.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
42818	The operands of an operator or function are not compatible.	The length of the pattern is longer than 4000 bytes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY009	Invalid argument value.	The pointer to the *LocatedAt* argument was NULL. The argument value for *FromPosition* was not greater than 0. *LocatorCType* is not one of SQL_C_CLOB_LOCATOR, SQL_C_BLOB_LOCATOR, or SQL_C_DBCLOB_LOCATOR.
HY010	Function sequence error.	The specified *StatementHandle* is not in an *allocated* state. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Table 116. SQLGetPosition SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY090	Invalid string or buffer length.	The value of *SearchLiteralLength* was less than 1, and not SQL_NTS.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.
0F001	The LOB token variable does not currently represent any value.	The value specified for *Locator* or *SearchLocator* is not currently a LOB locator.

Restrictions

This function is not available when connected to a DB2 server that does not support Large Objects. Call SQLGetFunctions() with the function type set to SQL_API_SQLGETPOSITION and check the *fExists* output argument to determine if the function is supported for the current connection.

Example

```
/* From CLI sample lookres.c */
/* ... */
    SQLCHAR * stmt2 = "SELECT resume FROM emp_resume "
                      "WHERE empno = ?  AND resume_format = 'ascii'" ;
/* ... */
    /* Get CLOB locator to selected Resume */

    rc = SQLBindParameter( hstmt,
                           1,
                           SQL_PARAM_OUTPUT,
                           emp_no.type,
                           SQL_CHAR,
                           emp_no.length,
                           0,
                           emp_no.s,
                           emp_no.length,
                           &emp_no.ind
                         ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf( "\n>Enter an employee number:\n" ) ;
    gets( ( char * ) emp_no.s ) ;

    rc = SQLExecDirect( hstmt, stmt2, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol( hstmt,
                     1,
                     SQL_C_CLOB_LOCATOR,
                     &ClobLoc1,
                     0,
                     &pcbValue
```

```
              ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLFetch( hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/*
 Search CLOB locator to find "Interests"
 Get substring of resume ( from position of interests to end )
*/

rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &lhstmt ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

/* Get total length */
rc = SQLGetLength( lhstmt,
                   SQL_C_CLOB_LOCATOR,
                   ClobLoc1,
                   &SLength,
                   &Ind ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, lhstmt, rc ) ;

/* Get Starting postion */
rc = SQLGetPosition( lhstmt,
                     SQL_C_CLOB_LOCATOR,
                     ClobLoc1,
                     0,
                     ( SQLCHAR * ) "Interests",
                     9,
                     1,
                     &Pos1,
                     &Ind
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, lhstmt, rc ) ;

rc = SQLFreeStmt( lhstmt, SQL_CLOSE ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, lhstmt, rc ) ;

buffer = ( SQLCHAR * ) malloc( SLength - Pos1 + 1 ) ;

/* Get just the "Interests" section of the Resume CLOB */
/* ( From Pos1 to end of CLOB ) */
rc = SQLGetSubString( lhstmt,
                      SQL_C_CLOB_LOCATOR,
                      ClobLoc1,
                      Pos1,
                      SLength - Pos1,
                      SQL_C_CHAR,
                      buffer,
                      SLength - Pos1 + 1,
                      &OutLength,
                      &Ind
                    ) ;
```

```
CHECK_HANDLE( SQL_HANDLE_STMT, lhstmt, rc ) ;
/* Print Interest section of Employee's resume */
printf( "\nEmployee #: %s\n %s\n", emp_no.s, buffer ) ;
```

References

- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)” on page 381
- “SQLFetch - Fetch Next Row” on page 396
- “SQLGetLength - Retrieve Length of A String Value” on page 532
- “SQLGetSubString - Retrieve Portion of A String Value” on page 550

SQLGetSQLCA - Get SQLCA Data Structure

Purpose

Specification:	DB2 CLI 2.1		

SQLGetSQLCA() is used to return the SQLCA associated with the preparation, and execution of an SQL statement, fetching of data, or the closing of a cursor. The SQLCA may return information in addition to what is available using SQLError().

Note: SQLGetSQLCA() must not be used as a replacement for SQLGetDiagField() and SQLGetDiagRec().

For a detailed description of the SQLCA structure, refer to the SQLCA appendix in the*SQL Reference*

An SQLCA is not available if a function is processed strictly on the application side, such as allocating a statement handle. In this case, an empty SQLCA is returned with all values set to zero.

Syntax

```
SQLRETURN   SQLGetSQLCA      (SQLHENV           EnvironmentHandle, /* henv */
                              SQLHDBC           ConnectionHandle,  /* hdbc */
                              SQLHSTMT          StatementHandle,   /* hstmt */
                              struct sqlca FAR  *SqlcaPtr);        /* pSqlca */
```

Function Arguments

Table 117. SQLGetSQLCA Arguments

Data Type	Argument	Use	Description
SQLHENV	EnvironmentHandle	input	Environment Handle. To obtain the SQLCA associated with an environment, pass a valid environment handle. Set *ConnectionHandle* and *StatementHandle* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
SQLHDBC	ConnectionHandle	input	Connection Handle. To obtain the SQLCA associated with a connection, pass a valid database connection handle, and set *StatementHandle* to SQL_NULL_HSTMT. The *EnvironmentHandle* argument is ignored.
SQLHSTMT	StatementHandle	input	Statement Handle. To obtain the SQLCA associated with a statement, pass a valid statement handle. The *EnvironmentHandle* and *ConnectionHandle* arguments are ignored.
SQLCA *	sqlCA	output	SQL Communication Area

Usage

The handles are used in the same way as for the SQLError() function. To obtain the SQLCA associated with:

- An environment, pass a valid environment handle. Set *ConnectionHandle* and *StatementHandle* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
- A connection, pass a valid database connection handle, and set *StatementHandle* to SQL_NULL_HSTMT. The *EnvironmentHandle* argument is ignored.
- A statement, pass a valid statement handle. The *EnvironmentHandle* and *ConnectionHandle* arguments are ignored.

If diagnostic information generated by one DB2 CLI function is not retrieved before a function other than SQLError() is called with the same handle, the information for the previous function call is lost. This is true whether or not diagnostic information is generated for the second DB2 CLI function call.

If a DB2 CLI function is called that does not result in interaction with the DBMS, then the SQLCA will contain all zeroes. Meaningful information will usually be returned for the following functions:

- SQLBrowseConnect()
- SQLCancel(),
- SQLCloseCursor()
- SQLColAttribute()
- SQLColumnPrivileges()
- SQLColumns()
- SQLConnect(), SQLDisconnect()
- SQLCopyDesc()
- SQLDataSources()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLEndTran()
- SQLExecDirect(), SQLExecute()
- SQLFetch()
- SQLFetchScroll()
- SQLForeignKeys()
- SQLFreeHandle()
- SQLGetData() (if LOB column is involved)
- SQLMoreResults()

- SQLPrepare()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLRowCount()
- SQLSetConnectAttr() (for SQL_ATTR_AUTOCOMMIT, and SQL_ATTR_DB2EXPLAIN)
- SQLStatistics()
- SQLTables()
- SQLTablePrivileges()
- SQLTransact()

If the database connection is to a DB2 Universal Database Version 2 server or later, there are two fields in the SQLCA that may be of particular interest:

- The SQLERRD(3) field (example, sqlca.errd[2]):
  - After PREPARE, contains an estimate of the number of rows that will be returned to the user when the statement is executed. An application can inform the user of this information to help assess whether the appropriate query has been issued.
  - After INSERT, DELETE, UPDATE, contains the actual number of rows affected.
  - After Compound SQL processing, contains an accumulation of all sub-statement rows affected by INSERT, UPDATE or DELETE statements.
- The SQLERRD(4) field (example, sqlca.errd[3]):
  - After a PREPARE, contains a relative cost estimate of the resources required to process the statement.

    This is the number that is compared to the DB2ESTIMATE configuration keyword as described in "Configuration Keywords" on page 164, and the SQL_ATTR_DB2ESTIMATE connection attribute as described in the function description, "SQLSetConnectAttr - Set Connection Attributes" on page 618.
  - After Compound SQL processing, contains a count of the number of successful sub-statements.

Note: The accuracy of the information returned in the SQLERRD(3) and SQLERRD(4) fields is dependent on many factors such as the use of parameter markers and expressions within the statement. The main factor which can be controlled is the accuracy of the database statistics. That is, when the statistics were last updated, (for example, for DB2 Universal Database, the last time the RUNSTATS command was run.)

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

None.

Restrictions

None.

Example

```
/* From CLI sample getsqlca.c */
/* ... */
    /* execute the SQL statement in "sqlstr"    */

    rc = SQLPrepare(hstmt, sqlstr, SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf(" Relative Cost=[%ld] Estimated # rows=[%ld]\n"
           " Continue with execution(Y or N)?\n",
           sqlca.sqlerrd[3], sqlca.sqlerrd[2]);
    gets((char *)prompt);
    if (prompt[0] == 'n' || prompt[0] =='N')
        return(0);

    if ( rc != SQL_SUCCESS )
       CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLExecute(hstmt);
```

References

- “SQLGetDiagRec - Get Multiple Fields Settings of Diagnostic Record” on page 477

SQLGetStmtAttr - Get Current Setting of a Statement Attribute

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLGetStmtAttr() returns the current setting of a statement attribute.

Syntax

```
SQLRETURN   SQLGetStmtAttr   (SQLHSTMT          StatementHandle,
                              SQLINTEGER        Attribute,
                              SQLPOINTER        ValuePtr,
                              SQLINTEGER        BufferLength,
                              SQLINTEGER        *StringLengthPtr);
```

Function Arguments

Table 118. SQLGetStmtAttr Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLINTEGER	*Attribute*	input	Attribute to retrieve.
SQLPOINTER	ValuePtr	output	Pointer to a buffer in which to return the value of the attribute specified in *Attribute*.

Table 118. SQLGetStmtAttr Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER	BufferLength	input	If *Attribute* is an ODBC-defined attribute and *ValuePtr* points to a character string or a binary buffer, this argument should be the length of **ValuePtr*. If *Attribute* is an ODBC-defined attribute and **ValuePtr* is an integer, *BufferLength* is ignored. If *Attribute* is a DB2 CLI attribute, the application indicates the nature of the attribute by setting the *BufferLength* argument. *BufferLength* can have the following values: • If **ValuePtr* is a pointer to a character string, then *BufferLength* is the length of the string or SQL_NTS. • If **ValuePtr* is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in *BufferLength*. This places a negative value in *BufferLength*. • If **ValuePtr* is a pointer to a value other than a character string or binary string, then *BufferLength* should have the value SQL_IS_POINTER. • If **ValuePtr* is contains a fixed-length data type, then *BufferLength* is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.
SQLSMALLINT	*StringLengthPtr	output	A pointer to a buffer in which to return the total number of bytes (excluding the null termination character) available to return in **ValuePtr*. If this is a null pointer, no length is returned. If the attribute value is a character string, and the number of bytes available to return is greater than or equal to *BufferLength*, the data in **ValuePtr* is truncated to *BufferLength* minus the length of a null termination character and is null terminated by the DB2 CLI.

Usage

A call to `SQLGetStmtAttr()` returns in **ValuePtr* the value of the statement attribute specified in *Attribute*. That value can either be a 32-bit value or a null-terminated character string. If the value is a null-terminated string, the application specifies the maximum length of that string in the *BufferLength*

argument, and DB2 CLI returns the length of that string in the **StringLengthPtrPtr* buffer. If the value is a 32-bit value, the *BufferLength* and *StringLengthPtr* arguments are not used.

In order to allow DB2 CLI Version 5.2 applications calling `SQLGetStmtAttr()` to work with DB2 CLI Version 2, a call to `SQLGetStmtAttr()` is mapped to `SQLGetStmtOption()`.

The following statement attributes are read-only, so can be retrieved by `SQLGetStmtAttr()`, but not set by `SQLSetStmtAttr()`. For a list of attributes that can be set and retrieved, see "SQLSetStmtAttr - Set Options Related to a Statement" on page 702.

- SQL_ATTR_IMP_PARAM_DESC
- SQL_ATTR_IMP_ROW_DESC
- SQL_ATTR_ROW_NUMBER

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 119. SQLGetStmtAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The data returned in **ValuePtr* was truncated to be *BufferLength* minus the length of a null termination character. The length of the untruncated string value is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state.	The argument *Attribute* was SQL_ATTR_ROW_NUMBER and the cursor was not open, or the cursor was positioned before the start of the result set or after the end of the result set.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.

Table 119. SQLGetStmtAttr SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	An asynchronously executing function was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()` or `SQLExecDirect()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument *BufferLength* was less than 0.
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid for this version of DB2 CLI
HY109	Invalid cursor position.	The *Attribute* argument was SQL_ATTR_ROW_NUMBER and the the row had been deleted or could not be fetched.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid DB2 CLI attribute for the version of DB2 CLI, but was not supported by the data source.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- “SQLGetConnectAttr - Get Current Attribute Setting” on page 439
- “SQLSetConnectAttr - Set Connection Attributes” on page 618
- “SQLSetStmtAttr - Set Options Related to a Statement” on page 702

SQLGetStmtOption - Return Current Setting of A Statement Option

Deprecated

Note:

In ODBC version 3, SQLGetStmtOption() has been deprecated and replaced with SQLGetStmtAttr(); see "SQLGetStmtAttr - Get Current Setting of a Statement Attribute" on page 545 for more information.

Although this version of DB2 CLI continues to support SQLGetStmtOption(), we recommend that you begin using SQLGetStmtAttr() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

SQLGetSubString - Retrieve Portion of A String Value

Purpose

Specification:	DB2 CLI 2.1		

SQLGetSubString() is used to retrieve a portion of a large object value, referenced by a large object locator that has been returned from the server (returned by a fetch or a previous SQLGetSubString() call) during the current transaction.

Syntax

```
SQLRETURN  SQLGetSubString   (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLSMALLINT       LocatorCType,
                SQLINTEGER        SourceLocator,
                SQLUINTEGER       FromPosition,
                SQLUINTEGER       ForLength,
                SQLSMALLINT       TargetCType,
                SQLPOINTER        DataPtr,           /* rgbValue */
                SQLINTEGER        BufferLength,      /* cbValueMax */
                SQLINTEGER   FAR  *StringLength,
                SQLINTEGER   FAR  *IndicatorValue);
```

Function Arguments

Table 120. SQLGetSubString Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle. This can be any statement handle which has been allocated but which does not currently have a prepared statement assigned to it.
SQLSMALLINT	LocatorCType	input	The C type of the source LOB locator. This may be: • SQL_C_BLOB_LOCATOR • SQL_C_CLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR
SQLINTEGER	Locator	input	*Locator* must be set to the source LOB locator value.
SQLUINTEGER	FromPosition	input	For BLOBs and CLOBs, this is the position of the first byte to be returned by the function. For DBCLOBs, this is the first character. The start byte or character is numbered 1.

Table 120. SQLGetSubString Arguments (continued)

Data Type	Argument	Use	Description
SQLUINTEGER	ForLength	input	This is the length of the string to be returned by the function. For BLOBs and CLOBs, this is the length in bytes. For DBCLOBs, this is the length in characters. If *FromPosition* is less than the length of the source string but *FromPosition* + *ForLength* - 1 extends beyond the end of the source string, the result is padded on the right with the necessary number of characters (X'00' for BLOBs, single byte blank character for CLOBs, and double byte blank character for DBCLOBs).
SQLSMALLINT	TargetCType	input	The C data type of the *DataPtr*. The target must always be either a LOB locator C buffer type (SQL_C_CLOB_LOCATOR, (SQL_C_BLOB_LOCATOR, (SQL_C_DBCLOB_LOCATOR) or a C string variable (SQL_C_CHAR for CLOB, SQL_C_BINARY for BLOB, and SQL_C_DBCHAR for DBCLOB).
SQLPOINTER	DataPtr	output	Pointer to the buffer where the retrieved string value or a LOB locator is to be stored.
SQLINTEGER	BufferLength	input	Maximum size of the buffer pointed to by *DataPtr* in bytes.
SQLINTEGER *	StringLength	output	The length of the returned information in *DataPtr* in bytes[a] if the target C buffer type is intended for a binary or character string variable and not a locator value. If the pointer is set to NULL, nothing is returned.
SQLINTEGER *	IndicatorValue	output	Always set to zero.

Note:

a This is in bytes even for DBCLOB data.

Usage

`SQLGetSubString()` is used to obtain any portion of the string that is represented by the LOB locator. There are two choices for the target:

- The target can be an appropriate C string variable.

- A new LOB value can be created on the server and the LOB locator for that value can be assigned to a target application variable on the client.

SQLGetSubString() can be used as an alternative to SQLGetData for getting data in pieces. In this case a column is first bound to a LOB locator, which is then used to fetch the LOB as a whole or in pieces.

The Locator argument can contain any valid LOB locator which has not been explicitly freed using a FREE LOCATOR statement nor implicitly freed because the transaction during which it was created has terminated.

The statement handle must not have been associated with any prepared statements or catalog function calls.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 121. SQLGetSubString SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The amount of data to be returned is longer than *BufferLength*. Actual length available for return is stored in *StringLength*.
07006	Invalid conversion.	The value specified for *TargetCType* was not SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR or a LOB locator. The value specified for *TargetCType* is inappropriate for the source (for example SQL_C_DBCHAR for a BLOB column).
22011	A substring error occurred.	*FromPosition* is greater than the of length of the source string.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	*LocatorCType* is not one of SQL_C_CLOB_LOCATOR, SQL_C_BLOB_LOCATOR, or SQL_C_DBCLOB_LOCATOR.

Table 121. SQLGetSubString SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY009	Invalid argument value.	The value specified for *FromPosition* or for *ForLength* was not a positive integer.
HY010	Function sequence error.	The specified *StatementHandle* is not in an *allocated* state. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value of *BufferLength* was less than 0.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.
0F001	No locator currently assigned	The value specified for *Locator* is not currently a LOB locator.

Restrictions

This function is not available when connected to a DB2 server that does not support Large Objects. Call `SQLGetFunctions()` with the function type set to SQL_API_SQLGETSUBSTRING and check the *fExists* output argument to determine if the function is supported for the current connection.

Example

Refer to “Example” on page 538.

References

- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408
- “SQLFetch - Fetch Next Row” on page 396
- “SQLGetData - Get Data From a Column” on page 447

SQLGetSubString

- “SQLGetLength - Retrieve Length of A String Value” on page 532
- “SQLGetPosition - Return Starting Position of String” on page 535

SQLGetTypeInfo - Get Data Type Information

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLGetTypeInfo() returns information about the data types that are supported by the DBMSs associated with DB2 CLI. The information is returned in an SQL result set. The columns can be received using the same functions that are used to process a query.

Syntax

```
SQLRETURN   SQLGetTypeInfo   (SQLHSTMT          StatementHandle,
                              SQLSMALLINT       DataType);
```

Function Arguments

Table 122. SQLGetTypeInfo Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.

Table 122. SQLGetTypeInfo Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	*DataType*	input	The SQL data type being queried. The supported types are: • SQL_ALL_TYPES • SQL_BIGINT • SQL_BINARY • SQL_BLOB • SQL_CHAR • SQL_CLOB • SQL_DATE • SQL_DBCLOB • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TIME • SQL_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC If SQL_ALL_TYPES is specified, information about all supported data types would be returned in ascending order by TYPE_NAME. All unsupported data types would be absent from the result set.

Usage

Since `SQLGetTypeInfo()` generates a result set and is equivalent to executing a query, it will generate a cursor and begin a transaction. To prepare and execute another statement on this statement handle, the cursor must be closed.

If SQLGetTypeInfo() is called with an invalid *DataType*, an empty result set is returned.

If either the LONGDATACOMPAT keyword or the SQL_ATTR_LONGDATA_COMPAT connection attribute is set, then SQL_LONGVARBINARY, SQL_LONGVARCHAR and SQL_LONGVARGRAPHIC will be returned for the *DATA_TYPE* argument instead of SQL_BLOB, SQL_CLOB and SQL_DBCLOB.

The columns of the result set generated by this function are described below.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change. The data types returned are those that can be used in a CREATE TABLE, ALTER TABLE, DDL statement. Non-persistent data types such as the locator data types are not part of the returned result set. User defined data types are not returned either.

Columns Returned by SQLGetTypeInfo

Column 1 TYPE_NAME (VARCHAR(128) NOT NULL Data Type)
Character representation of the SQL data type name, e.g. VARCHAR, BLOB, DATE, INTEGER

Column 2 DATA_TYPE (SMALLINT NOT NULL Data Type)
SQL data type define values, e.g. SQL_VARCHAR, SQL_BLOB, SQL_DATE, SQL_INTEGER.

Column 3 COLUMN_SIZE (INTEGER Data Type)
If the data type is a character or binary string, then this column contains the maximum length in bytes; if it is a graphic (DBCS) string, this is the number of double byte characters for the column.

For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character.

For numeric data types, this is the total number of digits.

Column 4 LITERAL_PREFIX (VARCHAR(128) Data Type)
Character that DB2 recognizes as a prefix for a literal of this data type. This column is null for data types where a literal prefix is not applicable.

Column 5 LITERAL_SUFFIX (VARCHAR(128) Data Type)
Character that DB2 recognizes as a suffix for a literal of this data type. This column is null for data types where a literal prefix is not applicable.

Column 6 CREATE_PARAMS (VARCHAR(128) Data Type)
The text of this column contains a list of keywords, separated by

commas, corresponding to each parameter the application may specify in parenthesis when using the name in the TYPE_NAME column as a data type in SQL. The keywords in the list can be any of the following: LENGTH, PRECISION, SCALE. They appear in the order that the SQL syntax requires that they be used.

A NULL indicator is returned if there are no parameters for the data type definition, (such as INTEGER).

Note: The intent of CREATE_PARAMS is to enable an application to customize the interface for a *DDL builder*. An application should expect, using this, only to be able to determine the number of arguments required to define the data type and to have localized text that could be used to label an edit control.

Column 7 NULLABLE (SMALLINT NOT NULL Data Type)
Indicates whether the data type accepts a NULL value

- Set to SQL_NO_NULLS if NULL values are disallowed.
- Set to SQL_NULLABLE if NULL values are allowed.

Column 8 CASE_SENSITIVE (SMALLINT NOT NULL Data Type)
Indicates whether the data type can be treated as case sensitive for collation purposes; valid values are SQL_TRUE and SQL_FALSE.

Column 9 SEARCHABLE (SMALLINT NOT NULL Data Type)
Indicates how the data type is used in a WHERE clause. Valid values are:

- SQL_UNSEARCHABLE : if the data type cannot be used in a WHERE clause.
- SQL_LIKE_ONLY : if the data type can be used in a WHERE clause only with the LIKE predicate.
- SQL_ALL_EXCEPT_LIKE : if the data type can be used in a WHERE clause with all comparison operators except LIKE.
- SQL_SEARCHABLE : if the data type can be used in a WHERE clause with any comparison operator.

Column 10 UNSIGNED_ATTRIBUTE (SMALLINT Data Type)
Indicates where the data type is unsigned. The valid values are: SQL_TRUE, SQL_FALSE or NULL. A NULL indicator is returned if this attribute is not applicable to the data type.

Column 11 FIXED_PREC_SCALE (SMALLINT NOT NULL Data Type)
Contains the value SQL_TRUE if the data type is exact numeric and always has the same precision and scale; otherwise, it contains SQL_FALSE.

Column 12 AUTO_INCREMENT (SMALLINT Data Type)
Contains SQL_TRUE if a column of this data type is automatically set to a unique value when a row is inserted; otherwise, contains SQL_FALSE.

Column 13 LOCAL_TYPE_NAME (VARCHAR(128) Data Type)
This column contains any localized (native language) name for the data type that is different from the regular name of the data type. If there is no localized name, this column is NULL.

This column is intended for display only. The character set of the string is locale-dependent and is typically the default character set of the database.

Column 14 MINIMUM_SCALE (INTEGER Data Type)
The minimum scale of the SQL data type. If a data type has a fixed scale, the MINIMUM_SCALE and MAXIMUM_SCALE columns both contain the same value. NULL is returned where scale is not applicable.

Column 15 MAXIMUM_SCALE (INTEGER Data Type)
The maximum scale of the SQL data type. NULL is returned where scale is not applicable. If the maximum scale is not defined separately in the DBMS, but is defined instead to be the same as the maximum length of the column, then this column contains the same value as the COLUMN_SIZE column.

Column 16 SQL_DATA_TYPE (SMALLINT NOT NULL Data Type)
The value of the SQL data type as it appears in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column (except for interval and datetime data types which DB2 CLI does not support).

Column 17 SQL_DATETIME_SUB (SMALLINT Data Type)
This field is always NULL (DB2 CLI does not support interval and datetime data types).

Column 18 NUM_PREC_RADIX (INTEGER Data Type)
If the data type is an approximate numeric type, this column contains the value 2 to indicate that COLUMN_SIZE specifies a number of bits. For exact numeric types, this column contains the value 10 to indicate that COLUMN_SIZE specifies a number of decimal digits. Otherwise, this column is NULL.

Column 19 INTERVAL_PRECISION (SMALLINT Data Type)
This field is always NULL (DB2 CLI does not support interval data types).

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 123. SQLGetTypeInfo SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle. *StatementHandle* had not been closed.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY004	SQL data type out of range.	An invalid *DataType* was specified.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Restrictions

The following ODBC specified SQL data types (and their corresponding *DataType* define values) are not supported by any IBM RDBMS:

Data Type	DataType
TINY INT	SQL_TINYINT
BIG INT	SQL_BIGINT
BIT	SQL_BIT

Example

```
/* From CLI sample typeinfo.c */
/* ... */
    rc = SQLGetTypeInfo(hstmt, SQL_ALL_TYPES);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) typename.s,
```

```
                128, &typename.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 2, SQL_C_DEFAULT, (SQLPOINTER) & datatype,
                sizeof(datatype), &datatype_ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 3, SQL_C_DEFAULT, (SQLPOINTER) & precision,
                sizeof(precision), &precision_ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 7, SQL_C_DEFAULT, (SQLPOINTER) & nullable,
                sizeof(nullable), &nullable_ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 8, SQL_C_DEFAULT, (SQLPOINTER) & casesens,
                sizeof(casesens), &casesens_ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

printf("Datatype              Datatype Precision  Nullable   Case\n");
printf("Typename               (int)                       Sensitive\n");
printf("---------------------- -------- ---------- -------- ---------\n");
/* LONG VARCHAR FOR BIT DATA  99  2147483647   FALSE    FALSE */
/* Fetch each row, and display */
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
    printf("%-25s ", typename.s);
    printf("%8d ", datatype);
    printf("%10ld ", precision);
    printf("%-8s ", truefalse[nullable]);
    printf("%-9s\n", truefalse[casesens]);
}                          /* endwhile */

if ( rc != SQL_NO_DATA_FOUND )
   CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

References

- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLSetColAttributes - Set Column Attributes” on page 617
- “SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns” on page 408
- “SQLGetInfo - Get General Information” on page 489

SQLMoreResults - Determine If There Are More Result Sets

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLMoreResults() determines whether there is more information available on the statement handle which has been associated with:

- array input of parameter values for a query, or
- a stored procedure that is returning result sets.

Syntax

```
SQLRETURN   SQLMoreResults   (SQLHSTMT          StatementHandle);  /* hstmt */
```

Function Arguments

Table 124. SQLMoreResults Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.

Usage

This function is used to return multiple results set in a sequential manner upon the execution of:

- a parameterized query with an array of input parameter values specified with SQLParamOptions() and SQLBindParameter(), or
- a stored procedure containing SQL queries, the cursors of which have been left open so that the result sets remain accessible when the stored procedure has finished execution.

Refer to “Using Arrays to Input Parameter Values” on page 83 and “Returning Result Sets from Stored Procedures” on page 128 for more information.

After completely processing the first result set, the application can call SQLMoreResults() to determine if another result set is available. If the current result set has unfetched rows, SQLMoreResults() discards them by closing the cursor and, if another result set is available, returns SQL_SUCCESS.

If all the result sets have been processed, SQLMoreResults() returns SQL_NO_DATA_FOUND.

If SQLFreeStmt() is called with the SQL_CLOSE option, or SQLFreeHandle() is called with *HandleType* set to SQL_HANDLE_STMT, all pending result sets on this statement handle are discarded.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

Diagnostics

Table 125. SQLMoreResults SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

In addition `SQLMoreResults()` can return the SQLSTATEs associated with `SQLExecute()`.

Restrictions

The ODBC specification of `SQLMoreResults()` also allow counts associated with the execution of parameterized INSERT, UPDATE, and DELETE statements with arrays of input parameter values to be returned. However, DB2 CLI does not support the return of such count information.

Example

```
/* From CLI sample ordrep.c */
/* ... */

    SQLCHAR * stmt =
       /* Common Table expression (or Define Inline View) */
       "WITH order (ord_num, cust_num, prod_num, quantity, amount) AS ( "
          "SELECT c.ord_num, c.cust_num, l.prod_num, l.quantity, "
                 "price(char(p.price, '.'), p.units, char(l.quantity, '.')) "
          "FROM ord_cust c, ord_line l, product p "
          "WHERE c.ord_num = l.ord_num "
                "AND l.prod_num = p.prod_num "
                "AND cast (cust_num as integer) = ? "
       "), "
       "totals (ord_num, total) AS ( "
          "SELECT ord_num, sum(decimal(amount, 10, 2)) "
          "FROM order GROUP BY ord_num "
       ") "
       /* The 'actual' SELECT from the inline view */
       "SELECT order.ord_num, cust_num, prod_num, quantity, "
              "DECIMAL(amount,10,2) amount, total "
       "FROM order, totals "
       "WHERE order.ord_num = totals.ord_num" ;

    /* Array of customers to get list of all orders for */
    SQLINTEGER Cust[] = {
        10,  20,  30,  40,  50,  60,  70,  80,  90, 100,
       110, 120, 130, 140, 150, 160, 170, 180, 190, 200,
       210, 220, 230, 240, 250,
    } ;

    /* Row-Wise (Includes buffer for both column data and length) */
    typedef struct {
       SQLINTEGER Ord_Num_L ;
       SQLINTEGER Ord_Num ;
       SQLINTEGER Cust_Num_L ;
       SQLINTEGER Cust_Num ;
       SQLINTEGER Prod_Num_L ;
       SQLINTEGER Prod_Num ;
       SQLINTEGER Quant_L ;
       SQLDOUBLE  Quant ;
       SQLINTEGER Amount_L ;
       SQLDOUBLE  Amount ;
       SQLINTEGER Total_L ;
       SQLDOUBLE  Total ;
    }  ord_info ;
    ord_info ord_array[row_array_size] ;

    SQLUINTEGER num_rows_fetched ;
    SQLUSMALLINT row_status_array[row_array_size], i, j ;

/* ... */
    /* Get details and total for each order Row-Wise */
    rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;
```

```
rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_PARAMSET_SIZE,
                     ( SQLPOINTER ) row_array_size,
                     0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindParameter( hstmt,
                       1,
                       SQL_PARAM_INPUT,
                       SQL_C_LONG,
                       SQL_INTEGER,
                       0,
                       0,
                       Cust,
                       0,
                       NULL
                     ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLExecDirect( hstmt, stmt, SQL_NTS ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* SQL_ROWSET_SIZE sets the max number of result rows to fetch each time */
rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_ROW_ARRAY_SIZE,
                     ( SQLPOINTER ) row_set_size,
                     0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Set Size of One row, Used for Row-Wise Binding Only */
rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_ROW_BIND_TYPE,
                     ( SQLPOINTER ) sizeof( ord_info ) ,
                     0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_ROW_STATUS_PTR,
                     ( SQLPOINTER ) row_status_array,
                     0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_ROWS_FETCHED_PTR,
                     ( SQLPOINTER ) &num_rows_fetched,
                     0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Bind column 1 to the Ord_num Field of the first row in the array */
```

```
    rc = SQLBindCol( hstmt,
                     1,
                     SQL_C_LONG,
                     ( SQLPOINTER ) & ord_array[0].Ord_Num,
                     0,
                     &ord_array[0].Ord_Num_L
                   ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* Bind remaining columns ... */
/* ... */
    /*
     NOTE: This sample assumes that an order will never have more
           rows than row_set_size.  A check should be added below to call
           SQLExtendedFetch multiple times for each result set.
    */

    while ( SQLFetchScroll( hstmt, SQL_FETCH_NEXT, 0 ) != SQL_NO_DATA ) {
       printf( "************************************\n" ) ;
       printf( "Orders for Customer: %ld\n", ord_array[0].Cust_Num ) ;
       printf( "************************************\n" ) ;
       i = 0 ;
       while ( i < num_rows_fetched ) {
          if ( row_status_array[i] == SQL_ROW_SUCCESS ||
               row_status_array[i] == SQL_ROW_SUCCESS_WITH_INFO
             ) {
             printf( "\nOrder #: %ld\n", ord_array[i].Ord_Num ) ;
             printf( "     Product  Quantity         Price\n" ) ;
             printf( "     -------- ---------------- ------------\n" ) ;
             j = i ;
             while ( ord_array[j].Ord_Num == ord_array[i].Ord_Num ) {
                printf( "    %8ld %16.7lf %12.2lf\n",
                        ord_array[i].Prod_Num,
                        ord_array[i].Quant,
                        ord_array[i].Amount
                      ) ;
                i++ ;
                if ( i >= num_rows_fetched ) break ;
                if ( row_status_array[i] != SQL_ROW_SUCCESS )
                   if ( row_status_array[i] != SQL_ROW_SUCCESS_WITH_INFO )
                      break ;
             }
             printf( "                                 ============\n" ) ;
             printf( "                                %12.2lf\n",
                     ord_array[j].Total
                   ) ;
          }
          else i++ ;
       }
    }
```

References

- “SQLParamOptions - Specify an Input Array for a Parameter” on page 579

SQLNativeSql - Get Native SQL Text

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLNativeSql() is used to show how DB2 CLI interprets vendor escape clauses. If the original SQL string passed in by the application contained vendor escape clause sequences, then DB2 CLI will return the transformed SQL string that would be seen by the data source (with vendor escape clauses either converted or discarded, as appropriate).

Syntax

```
SQLRETURN   SQLNativeSql     (
                SQLHDBC           ConnectionHandle,  /* hdbc */
                SQLCHAR      FAR  *InStatementText,  /* szSqlStrIn */
                SQLINTEGER        TextLength1,       /* cbSqlStrIn */
                SQLCHAR      FAR  *OutStatementText, /* szSqlStr */
                SQLINTEGER        BufferLength,      /* cbSqlStrMax */
                SQLINTEGER   FAR  *TextLength2Ptr);  /* pcbSqlStr */
```

Function Arguments

Table 126. SQLNativeSql Arguments

Data Type	Argument	Use	Description
SQLHDBC	ConnectionHandle	input	Connection Handle
SQLCHAR *	InStatementText	input	Input SQL string
SQLINTEGER	TextLength1	input	Length of *InStatementText*
SQLCHAR *	OutStatementText	output	Pointer to buffer for the transformed output string
SQLINTEGER	BufferLength	input	Size of buffer pointed by *OutStatementText*
SQLINTEGER *	TextLength2Ptr	output	The total number of bytes (excluding the null-terminator) available to return in *OutStatementText*. If the number of bytes available to return is greater than or equal to *BufferLength*, the output SQL string in *OutStatementText* is truncated to *BufferLength - 1* bytes.

Usage

This function is called when the application wishes to examine or display the transformed SQL string that would be passed to the data source by DB2 CLI. Translation (mapping) would only occur if the input SQL statement string contains vendor escape clause sequence(s). For more information on vendor escape clause sequences, refer to “Using Vendor Escape Clauses” on page 144.

DB2 CLI can only detect vendor escape clause syntax errors; since DB2 CLI does not pass the transformed SQL string to the data source for preparation, syntax errors that are detected by the DBMS are not generated at this time. (The statement is not passed to the data source for preparation because the preparation may potentially cause the initiation of a transaction.)

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 127. SQLNativeSql SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer *OutStatementText* was not large enough to contain the entire SQL string, so truncation occurred. The argument *TextLength2Ptr* contains the total length of the untruncated SQL string. (Function returns with SQL_SUCCESS_WITH_INFO)
08003	Connection is closed.	The *ConnectionHandle* does not reference an open database connection.
37000	Invalid SQL syntax.	The input SQL string in *InStatementText* contained a syntax error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY009	Invalid argument value.	The argument *InStatementText* is a NULL pointer. The argument *OutStatementText* is a NULL pointer.
HY090	Invalid string or buffer length.	The argument *TextLength1* was less than 0, but not equal to SQL_NTS. The argument *BufferLength* was less than 0.

Restrictions

None.

Example

```
/* From CLI sample native.c */
/* ... */
    SQLCHAR in_stmt[1024], out_stmt[1024] ;
    SQLSMALLINT pcPar ;
    SQLINTEGER indicator ;
```

```
/* ... */
    /* Prompt for a statement to prepare */
    printf("Enter an SQL statement: \n");
    gets((char *)in_stmt);

    /* prepare the statement */
    rc = SQLPrepare(hstmt, in_stmt, SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    SQLNumParams(hstmt, &pcPar);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    SQLNativeSql(hstmt, in_stmt, SQL_NTS, out_stmt, 1024, &indicator);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    if ( indicator == SQL_NULL_DATA ) printf( "Invalid statement\n" ) ;
    else {
       printf( "Input Statement: \n %s \n", in_stmt ) ;
       printf( "Output Statement: \n %s \n", in_stmt ) ;
       printf( "Number of Parameter Markers = %d\n", pcPar ) ;
    }

    rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

References

- “Using Vendor Escape Clauses” on page 144

SQLNumParams - Get Number of Parameters in A SQL Statement

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLNumParams() returns the number of parameter markers in an SQL statement.

Syntax

```
SQLRETURN    SQLNumParams     (SQLHSTMT          StatementHandle,
                               SQLSMALLINT  FAR  *ParameterCountPtr);
```

Function Arguments

Table 128. SQLNumParams Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.
SQLSMALLINT *	ParameterCountPtr	Output	Number of parameters in the statement.

Usage

This function can only be called after the statement associated with *StatementHandle* has been prepared. If the statement does not contain any parameter markers, *ParameterCountPtr* is set to 0.

An application can call this function to determine how many SQLBindParameter() (or SQLBindFileToParam()) calls are necessary for the SQL statement associated with the statement handle.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 129. SQLNumParams SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.

Table 129. SQLNumParams SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	This function was called before `SQLPrepare()` was called for the specified *StatementHandle* The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Restrictions

None.

Example

Refer to “Example” on page 568.

References

- “SQLBindFileToParam - Bind LOB File Reference to LOB Parameter” on page 242
- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247
- “SQLPrepare - Prepare a Statement” on page 583

SQLNumResultCols - Get Number of Result Columns

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLNumResultCols() returns the number of columns in the result set associated with the input statement handle.

SQLPrepare() or SQLExecDirect() must be called before calling this function.

After calling this function, you can call SQLColAttribute(), or one of the bind column functions.

Syntax

```
SQLRETURN   SQLNumResultCols (SQLHSTMT          StatementHandle,   /* hstmt */
                              SQLSMALLINT  FAR  *ColumnCountPtr);  /* pccol */
```

Function Arguments

Table 130. SQLNumResultCols Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLSMALLINT *	*ColumnCountPtr*	output	Number of columns in the result set

Usage

The function sets the output argument to zero if the last statement or function executed on the input statement handle did not generate a result set.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 131. SQLNumResultCols SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.

Table 131. SQLNumResultCols SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The function was called prior to calling `SQLPrepare()` or `SQLExecDirect()` for the *StatementHandle*. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Authorization

None.

Example

Refer to “Example” on page 340.

References

- “SQLBindCol - Bind a Column to an Application Variable or LOB Locator” on page 227
- “SQLBindFileToCol - Bind LOB File Reference to LOB Column” on page 237
- “SQLSetColAttributes - Set Column Attributes” on page 617

SQLNumResultCols

- “SQLDescribeCol - Return a Set of Attributes for a Column” on page 336
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLGetData - Get Data From a Column” on page 447
- “SQLPrepare - Prepare a Statement” on page 583

SQLParamData - Get Next Parameter For Which A Data Value Is Needed

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	ISO CLI

SQLParamData() is used in conjunction with SQLPutData() to send long data in pieces. It can also be used to send fixed length data as well. For a description of the exact sequence of this input method, refer to "Sending/Retrieving Long Data in Pieces" on page 80.

Syntax

```
SQLRETURN   SQLParamData     (SQLHSTMT          StatementHandle,
                              SQLPOINTER   FAR  *ValuePtrPtr );
```

Function Arguments

Table 132. SQLParamData Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLPOINTER *	ValuePtrPtr	output	Pointer to a buffer in which to return the address of the *ParameterValuePtr* buffer specified in SQLBindParameter() (for parameter data) or the address of the *TargetValuePtr* buffer specified in SQLBindCol() (for column data), as contained in the SQL_DESC_DATA_PTR descriptor record field.

Usage

SQLParamData() returns SQL_NEED_DATA if there is at least one SQL_DATA_AT_EXEC parameter for which data still has not been assigned. This function returns an application provided value in *ValuePtrPtr* supplied by the application during the previous SQLBindParameter() call. SQLPutData() is called one or more times (in the case of long data) to send the parameter data. SQLParamData() is called to signal that all the data has been sent for the current parameter and to advance to the next SQL_DATA_AT_EXEC parameter. SQL_SUCCESS is returned when all the parameters have been assigned data values and the associated statement has been executed successfully. If any errors occur during or before actual statement execution, SQL_ERROR is returned.

If SQLParamData() returns SQL_NEED_DATA, then only SQLPutData() or SQLCancel() calls can be made. All other function calls using this statement handle will fail. In addition, all function calls referencing the parent *hdbc* of

StatementHandle will fail if they involve changing any attribute or state of that connection; that is, that following function calls on the parent *hdbc* are also not permitted:

- SQLAllocConnect()
- SQLAllocStmt()
- SQLSetConnectAttr()
- SQLNativeSql()
- SQLTransact()

Should they be invoked during an SQL_NEED_DATA sequence, these function will return SQL_ERROR with SQLSTATE of **HY**010 and the processing of the SQL_DATA_AT_EXEC parameters will not be affected.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA

Diagnostics

`SQLParamData()` can return any SQLSTATE returned by the `SQLExecDirect()` and `SQLExecute()` functions. In addition, the following diagnostics can also be generated:

Table 133. SQLParamData SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	Transfer of data between DB2 CLI and the application variables would result in incompatible data conversion.

Table 133. SQLParamData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22026	String data, length mismatch	The SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo() was 'Y' and less data was sent for a long parameter (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or other long data type) than was specified with the *StrLen_or_IndPtr* argument in SQLBindParameter(). The SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo() was 'Y' and less data was sent for a long column (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or other longdata type) than was specified in the length buffer corresponding to a column in a row of data that was updated with SQLSetPos().
40001	Transaction rollback.	The transaction to which this SQL statement belonged was rolled back due to a deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no specific SQLSTATE was defined. The error message returned by SQLError() in the argument *szErrorMsg* describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, SQLCancel() was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, SQLCancel() was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	SQLParamData() was called out of sequence. This call is only valid after an SQLExecDirect() or an SQLExecute(), or after an SQLPutData() call. Even though this function was called after an SQLExecDirect() or an SQLExecute() call, there were no SQL_DATA_AT_EXEC parameters (left) to process.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Table 133. SQLParamData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY092	Option type out of range.	The *FileOptions* argument of a previous SQLBindFileToParam() operation was not valid.
HY506	Error closing a file.	Error encountered while trying to close a temporary file.
HY509	Error deleting a file.	Error encountered while trying to delete a temporary file.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

None.

Example

Refer to “Example” on page 612.

References

- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247
- “SQLCancel - Cancel Statement” on page 290
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLPutData - Passing Data Value for A Parameter” on page 609
- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247

SQLParamOptions - Specify an Input Array for a Parameter

Usage

Note:

In ODBC version 3, SQLParamOptions() has been deprecated and replaced with SQLSetStmtAttr(); see "SQLSetStmtAttr - Set Options Related to a Statement" on page 702 for more information.

Although this version of DB2 CLI continues to support SQLParamOptions(), we recommend that you begin using SQLSetStmtAttr() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Purpose

Specification:	**DB2 CLI** 2.1	**ODBC** 1.0	

SQLParamOptions() provides the ability to set multiple values for each parameter set by SQLBindParameter(). This allows the application to perform batched processing of the same SQL statement with one set of prepare, execute and SQLBindParameter() calls.

Syntax

```
SQLRETURN   SQLParamOptions  (SQLHSTMT          StatementHandle,   /* hstmt */
                              SQLUINTEGER       Crow,              /* crow */
                              SQLUINTEGER  FAR  *FetchOffsetPtr);  /* pirow */
```

Function Arguments

Table 134. SQLParamOptions Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.
SQLUINTEGER	Crow	Input	Number of values for each parameter. If this is greater than 1, then the *rgbValue* argument in SQLBindParameter() points to an array of parameter values, and *pcbValue* points to an array of lengths.

Table 134. SQLParamOptions Arguments (continued)

Data Type	Argument	Use	Description
SQLUINTEGER *	FetchOffsetPtr	Output (deferred)	Pointer to the buffer for the current parameter array index. As each set of parameter values is processed, *FetchOffsetPtr* is set to the array index of that set. If a statement fails, *FetchOffsetPtr* can be used to determine how many statements were successfully processed. Nothing is returned if the *FetchOffsetPtr* pointer is NULL.

Usage

As a statement executes, *FetchOffsetPtr* is set to the index of the current array of parameter values. If an error occurs during execution for a particular element in the array, execution halts and `SQLExecute()`, `SQLExecDirect()` or `SQLParamData()` returns SQL_ERROR.

The contents of *FetchOffsetPtr* have the following uses:

- When `SQLParamData()` returns SQL_NEED_DATA, the application can access the value in *FetchOffsetPtr* to determine which set of parameters is being assigned values.
- When `SQLExecute()` or `SQLExecDirect()` returns an error, the application can access the value in *FetchOffsetPtr* to find out which element in the parameter value array failed.
- When `SQLExecute()`, `SQLExecDirect()`, `SQLParamData()`, or `SQLPutData()` succeeds, the value in *FetchOffsetPtr* is set to the input value in *Crow* to indicate that all elements of the array have been processed successfully.

The output argument *FetchOffsetPtr* indicates how many sets of parameters were successfully processed. If the statement processed is a query, *FetchOffsetPtr* indicates the array index associated with the current result set returned by `SQLMoreResults()` and is incremented each time `SQLMoreResults()` is called.

In environments where the underlying support allows Compound SQL (DB2 Universal Database, or DRDA environments with DB2 Connect V2.3), all the data in the array(s) together with the execute request are packaged together as one network flow.

When connected to DB2 Universal Database V2.1 or later, the application has the option of choosing ATOMIC or NOT ATOMIC Compound SQL. With ATOMIC Compound SQL (which is the default), either all the elements of the array are processed successfully, or none at all. With NOT ATOMIC Compound SQL, execution will continue even if an error is detected with one

of the intermediate array elements. The application can choose to select the type of Compound SQL by setting the SQL_ATTR_PARAMOPT_ATOMIC attribute of the SQLSetStmtAttr() call.

For DRDA environments, the underlying Compound SQL support is always NOT ATOMIC COMPOUND SQL (and therefore the default in DRDA scenarios).

If the application is not sure what the current value of the SQL_ATTR_PARAMOPT_ATOMIC attribute, it should call SQLGetStmtOption().

When connected to servers that do not support compound SQL, DB2 CLI prepares the statement, and executes it repeatedly for the array of parameter markers.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 135. SQLParamOptions SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY107	Row value out of range.	The value in the argument *Crow* was less than 1.

Restrictions

None.

Example

Refer to “Array Input Example” on page 88.

References

- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247
- “SQLMoreResults - Determine If There Are More Result Sets” on page 562
- “SQLSetStmtAttr - Set Options Related to a Statement” on page 702

SQLPrepare - Prepare a Statement

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the DBMS to be prepared. The application can reference this prepared statement by passing the statement handle to other functions.

If the statement handle has been previously used with a query statement (or any function that returns a result set), SQLFreeStmt() must be called to close the cursor, before calling SQLPrepare().

Syntax

```
SQLRETURN   SQLPrepare       (SQLHSTMT          StatementHandle,
                              SQLCHAR      FAR  *StatementText,
                              SQLINTEGER        TextLength);
```

Function Arguments

Table 136. SQLPrepare Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle. There must not be an open cursor associated with *StatementHandle*.
SQLCHAR *	*StatementText*	input	SQL statement string
SQLINTEGER	*TextLength*	input	Length of contents of *StatementText* argument. This must be set to either the exact length of the SQL statement in *szSqlstr*, or to SQL_NTS if the statement text is null-terminated.

Usage

If the SQL statement text contains vendor escape clause sequences, DB2 CLI will first modify the SQL statement text to the appropriate DB2 specific format before submitting it to the database for preparation. If the application does not generate SQL statements that contain vendor escape clause sequences (see “Using Vendor Escape Clauses” on page 144); then the SQL_ATTR_NOSCAN statement attribute should be set to SQL_NOSCAN at the connection level so that DB2 CLI does not perform a scan for any vendor escape clauses.

Once a statement has been prepared using SQLPrepare(), the application can request information about the format of the result set (if the statement was a query) by calling:

- SQLNumResultCols()
- SQLDescribeCol()
- SQLColAttribute()

The SQL statement string may contain parameter markers and SQLNumParams() can be called to determine the number of parameter markers in the statement. A parameter marker is represented by a "?" character, and is used to indicate a position in the statement where an application supplied value is to be substituted when SQLExecute() is called. The bind parameter functions, SQLBindParameter(), SQLSetParam() and SQLBindFileToParam() are used to bind (associate) application values with each parameter marker and to indicate if any data conversion should be performed at the time the data is transferred.

All parameters must be bound before calling SQLExecute(), for more information refer to "SQLExecute - Execute a Statement" on page 373.

Refer to the PREPARE section of the *SQL Reference* for information on rules related to parameter markers.

Once the application has processed the results from the SQLExecute() call, it can execute the statement again with new (or the same) parameter values.

The SQL statement cannot be a COMMIT or ROLLBACK. SQLTransact() must be called to issue COMMIT or ROLLBACK. For more information about supported SQL statements in DB2 Universal Database, refer to Table 215 on page 843 .

If the SQL statement is a Positioned DELETE or a Positioned UPDATE, the cursor referenced by the statement must be defined on a separate statement handle under the same connection handle and same isolation level.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 137. SQLPrepare SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	*StatementText* contained an UPDATE or DELETE statement which did not contain a WHERE clause.
01508	Statement disqualified for blocking.	The statement was disqualified for blocking for reasons other than storage.
21S01	Insert value list does not match column list.	*StatementText* contained an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*StatementText* contained a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
22018	Invalid character value for cast specification.	**StatementText* contained an SQL statement that contained a literal or parameter and the value was incompatible with the data type of the associated table column.
22019	Invalid escape character	The argument *StatementText* contained a LIKE predicate with an ESCAPE in the WHERE clause, and the length of the escape character following ESCAPE was not equal to 1.
22025	Invalid escape sequence	The argument *StatementText* contained "LIKE *pattern value* ESCAPE *escape character*" in the WHERE clause, and the character following the escape character in the pattern value was not one of "%" or "_".
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
34000	Invalid cursor name.	*StatementText* contained a Positioned DELETE or a Positioned UPDATE and the cursor referenced by the statement being executed was not open.
37xxx [a]	Invalid SQL syntax.	*StatementText* contained one or more of the following: • a COMMIT • a ROLLBACK • an SQL statement that the connected database server could not prepare • a statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belonged was rolled back due to deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.

SQLPrepare

Table 137. SQLPrepare SQLSTATEs (continued)

SQLSTATE	Description	Explanation
42xxx [a]	Syntax Error or Access Rule Violation.	**42**5xx indicates the authorization ID does not have permission to execute the SQL statement contained in *StatementText*. Other **42**xxx SQLSTATES indicate a variety of syntax or access problems with the statement.
58004	Unexpected system failure.	Unrecoverable system error.
S0001	Database object already exists.	*StatementText* contained a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already existed.
S0002	Database object does not exist.	*StatementText* contained an SQL statement that references a table name or a view name which did not exist.
S0011	Index already exists.	*StatementText* contained a CREATE INDEX statement and the specified index name already existed.
S0012	Index not found.	*StatementText* contained a DROP INDEX statement and the specified index name did not exist.
S0021	Column already exists.	*StatementText* contained an ALTER TABLE statement and the column specified in the ADD clause was not unique or identified an existing column in the base table.
S0022	Column not found.	*StatementText* contained an SQL statement that references a column name which did not exist.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY009	Invalid argument value.	*StatementText* was a null pointer.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Table 137. SQLPrepare SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The argument *TextLength* was less than 1, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Note:

a xxx refers to any SQLSTATE with that class code. Example, **37**xxx refers to any SQLSTATE in the **37** class.

Note: Not all DBMSs report all of the above diagnostic messages at prepare time. Therefore an application must also be able to handle these conditions when calling SQLExecute().

Authorization

None.

Example

```
/* From CLI sample prepare.c */
/* ... */
    SQLCHAR * sqlstmt =
       "SELECT deptname, location from org where division = ? " ;
/* ... */

    /* prepare statement for multiple use */
    rc = SQLPrepare(hstmt, sqlstmt, SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* bind division to parameter marker in sqlstmt */
    rc = SQLBindParameter( hstmt,
                           1,
                           SQL_PARAM_INPUT,
                           SQL_C_CHAR,
                           SQL_CHAR,
                           10,
                           0,
                           division.s,
                           11,
                           NULL
                         ) ;
```

```
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* bind deptname to first column in the result set */
rc = SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) deptname.s, 15,
                &deptname.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) location.s, 14,
                &location.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

printf("\nEnter Division Name or 'q' to quit:\n");
printf("(Eastern, Western, Midwest, Corporate)\n");
gets((char *)division.s);

while (division.s[0] != 'q') {

    rc = SQLExecute(hstmt);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf("Departments in %s Division:\n", division.s);
    printf("DEPTNAME       Location\n");
    printf("-------------- -------------\n");

    while ( ( rc = SQLFetch( hstmt ) ) == SQL_SUCCESS )
       printf( "%-14.14s %-13.13s \n", deptname.s, location.s ) ;
    if ( rc != SQL_NO_DATA_FOUND )
       CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLFreeStmt( hstmt, SQL_CLOSE ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf("\nEnter Division Name or 'q' to quit:\n");
    printf("(Eastern, Western, Midwest, Corporate)\n");
    gets((char *)division.s);
}

rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

References

- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247
- “SQLBindFileToParam - Bind LOB File Reference to LOB Parameter” on page 242
- “SQLSetColAttributes - Set Column Attributes” on page 617
- “SQLDescribeCol - Return a Set of Attributes for a Column” on page 336
- “SQLExecDirect - Execute a Statement Directly” on page 365
- “SQLExecute - Execute a Statement” on page 373

- “SQLNumParams - Get Number of Parameters in A SQL Statement” on page 570
- “SQLNumResultCols - Get Number of Result Columns” on page 572
- “SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator” on page 247

SQLPrimaryKeys - Get Primary Key Columns of A Table

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLPrimaryKeys() returns a list of column names that comprise the primary key for a table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLPrimaryKeys   (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLCHAR      FAR  *CatalogName,      /* szCatalogName */
                SQLSMALLINT       NameLength1,       /* cbCatalogName */
                SQLCHAR      FAR  *SchemaName,       /* szSchemaName */
                SQLSMALLINT       NameLength2,       /* cbSchemaName */
                SQLCHAR      FAR  *TableName,        /* szTableName */
                SQLSMALLINT       NameLength3);      /* cbTableName */
```

Function Arguments

Table 138. SQLPrimaryKeys Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLCHAR *	CatalogName	input	Catalog qualifier of a 3 part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	NameLength1	input	Length of *CatalogName*
SQLCHAR *	SchemaName	input	Schema qualifier of table name.
SQLSMALLINT	NameLength2	input	Length of *SchemaName*
SQLCHAR *	TableName	input	Table name.
SQLSMALLINT	NameLength3	input	Length of *TableName*

Usage

SQLPrimaryKeys() returns the primary key columns from a single table, Search patterns cannot be used to specify the schema qualifier or the table name.

The result set contains the columns listed in “Columns Returned By SQLPrimaryKeys” on page 591, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME and ORDINAL_POSITION.

Since calls to SQLPrimaryKeys() in many cases map to a complex and, thus, expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns Returned By SQLPrimaryKeys

Column 1 TABLE_CAT (VARCHAR(128))
This is always null.

Column 2 TABLE_SCHEM (VARCHAR(128))
The name of the schema containing TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL)
Name of the specified table.

Column 4 COLUMN_NAME (VARCHAR(128) not NULL)
Primary Key column name.

Column 5 ORDINAL_POSITION (SMALLINT not NULL)
Column sequence number in the primary key, starting with 1.

Column 6 PK_NAME (VARCHAR(128))
Primary key identifier. NULL if not applicable to the data source.

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the SQLPrimaryKeys() result set in ODBC.

If the specified table does not contain a primary key, an empty result set is returned.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 139. SQLPrimaryKeys SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal SQL_NTS.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetConnectAttr()`.

Restrictions

None.

Example

References

- “SQLForeignKeys - Get the List of Foreign Key Columns” on page 420
- “SQLStatistics - Get Index and Statistics Information For A Base Table” on page 733

SQLProcedureColumns - Get Input/Output Parameter Information for A Procedure

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLProcedureColumns() returns a list of input and output parameters associated with a procedure. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN SQLProcedureColumns(
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLCHAR      FAR  *CatalogName,      /* szProcCatalog */
                SQLSMALLINT       NameLength1,       /* cbProcCatalog */
                SQLCHAR      FAR  *SchemaName,       /* szProcSchema */
                SQLSMALLINT       NameLength2,       /* cbProcSchema */
                SQLCHAR      FAR  *ProcName,         /* szProcName */
                SQLSMALLINT       NameLength3,       /* cbProcName */
                SQLCHAR      FAR  *ColumnName,       /* szColumnName */
                SQLSMALLINT       NameLength4);      /* cbColumnName */
```

Function Arguments

Table 140. SQLProcedureColumns Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLCHAR *	CatalogName	input	Catalog qualifier of a 3 part procedure name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	NameLength1	input	Length of *CatalogName*. This must be set to 0.
SQLCHAR *	SchemaName	input	Buffer that may contain a *pattern-value* to qualify the result set by schema name. For DB2 for MVS/ESA V 4.1, all the stored procedures are in one schema; the only acceptable value for the *SchemaName* argument is a null pointer. For DB2 Universal Database, *SchemaName* can contain a valid pattern value. For more information about valid search patterns, refer to “Querying System Catalog Information” on page 65.
SQLSMALLINT	NameLength2	input	Length of *SchemaName*

Table 140. SQLProcedureColumns Arguments (continued)

Data Type	Argument	Use	Description
SQLCHAR *	ProcName	input	Buffer that may contain a *pattern-value* to qualify the result set by procedure name.
SQLSMALLINT	NameLength3	input	Length of *ProcName*
SQLCHAR *	ColumnName	input	Buffer that may contain a *pattern-value* to qualify the result set by parameter name. This argument is to be used to further qualify the result set already restricted by specifying a non-empty value for ProcName and/or SchemaName.
SQLSMALLINT	NameLength4	input	Length of *ColumnName*

Usage

DB2 Universal Database version 5 introduced two system catalog views used to store information about all stored procedures on the server (SYSCAT.PROCEDURES and SYSCAT.PROCPARMS). See "Appendix G. Catalog Views for Stored Procedures" on page 837 for information on these views.

Before version 5, DB2 CLI used the pseudo catalog table for stored procedure registration. By default, DB2 CLI will use the new system catalog views. If the application expects to use the pseudo catalog table then the CLI/ODBC configuration keyword PATCH1 should be set to 262144. See "Replacement of the Pseudo Catalog Table for Stored Procedures" on page 769 for more information.

If the stored procedure is at a DB2 for MVS/ESA V 4.1 server or later, the name of the stored procedures must be registered in the server's SYSIBM.SYSPROCEDURES catalog table.

For versions of other DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set will be returned.

DB2 CLI will return information on the input, input/output, and output parameters associated with the stored procedure, but cannot return information on the descriptor information for any result sets returned.

`SQLProcedureColumns()` returns the information in a result set, ordered by PROCEDURE_CAT, PROCEDURE_SCHEM, PROCEDURE_NAME, and COLUMN_TYPE. "Columns Returned By SQLProcedureColumns" on page 596 lists the columns in the result set. Applications should be aware that columns beyond the last column may be defined in future releases.

Since calls to SQLProcedureColumns() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

If the SQL_ATTR_LONGDATA_COMPAT connection attribute is set, LOB column types will be reported as LONG VARCHAR, LONG VARBINARY or LONG VARGRAPHIC types.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change. There were changes to these columns between version 2 and version 5. See “Changes to SQLProcedureColumns() Return Values” on page 772 for more information if you are running a version 2 DB2 CLI application that uses SQLProcedureColumns(), against a version 5 or later server.

Columns Returned By SQLProcedureColumns

Column 1 PROCEDURE_CAT (VARCHAR(128))
This is always null.

Column 2 PROCEDURE_SCHEM (VARCHAR(128))
The name of the schema containing PROCEDURE_NAME. (This is also NULL for DB2 for MVS/ESA V 4.1 SQLProcedureColumns() result sets.)

Column 3 PROCEDURE_NAME (VARCHAR(128))
Name of the procedure.

Column 4 COLUMN_NAME (VARCHAR(128))
Name of the parameter.

Column 5 COLUMN_TYPE (SMALLINT not NULL)
Identifies the type information associated with this row. The values can be:

- SQL_PARAM_TYPE_UNKNOWN : the parameter type is unknown.

  Note: This is not returned.
- SQL_PARAM_INPUT : this parameter is an input parameter.

- SQL_PARAM_INPUT_OUTPUT : this parameter is an input / output parameter.
- SQL_PARAM_OUTPUT : this parameter is an output parameter.
- SQL_RETURN_VALUE : the procedure column is the return value of the procedure.

  Note: This is not returned.
- SQL_RESULT_COL : this parameter is actually a column in the result set.

  Note: This is not returned.

Column 6 DATA_TYPE (SMALLINT not NULL)
SQL data type.

Column 7 TYPE_NAME (VARCHAR(128) not NULL)
Character string representing the name of the data type corresponding to DATA_TYPE.

Column 8 COLUMN_SIZE (INTEGER)
If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in bytes; if it is a graphic (DBCS) string, this is the number of double byte characters for the parameter.

For date, time, timestamp data types, this is the total number of bytes required to display the value when converted to character.

For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.

See also Table 194 on page 817.

Column 9 BUFFER_LENGTH (INTEGER)
The maximum number of bytes for the associated C buffer to store data from this parameter if SQL_C_DEFAULT were specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` calls. This length excludes any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.

See Table 196 on page 819.

Column 10 DECIMAL_DIGITS (SMALLINT)
The scale of the parameter. NULL is returned for data types where scale is not applicable.

See Table 195 on page 819.

Column 11 NUM_PREC_RADIX (SMALLINT)
Either 10 or 2 or NULL. If DATA_TYPE is an approximate numeric

data type, this column contains the value 2, then the COLUMN_SIZE column contains the number of bits allowed in the parameter.

If DATA_TYPE is an exact numeric data type, this column contains the value 10 and the COLUMN_SIZE and DECIMAL_DIGITS columns contain the number of decimal digits allowed for the parameter.

For numeric data types, the DBMS can return a NUM_PREC_RADIX of either 10 or 2.

NULL is returned for data types where radix is not applicable.

Column 12 NULLABLE (SMALLINT not NULL)
SQL_NO_NULLS if the parameter does not accept NULL values.

SQL_NULLABLE if the parameter accepts NULL values.

Column 13 REMARKS (VARCHAR(254))
May contain descriptive information about the parameter.

Column 14 COLUMN_DEF (VARCHAR)
The default value of the column.

If NULL was specified as the default value, then this column is the word NULL, not enclosed in quotation marks. If the default value cannot be represented without truncation, then this column contains TRUNCATED, with no enclosing single quotation marks. If no default value was specified, then this column is NULL.

The value of COLUMN_DEF can be used in generating a new column definition, except when it contains the value TRUNCATED.

Column 15 SQL_DATA_TYPE (SMALLINT not NULL)
The value of the SQL data type as it appears in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column except for datetime data types (DB2 CLI does not support interval data types).

For datetime data types, the SQL_DATA_TYPE field in the result set will be SQL_DATETIME, and the SQL_DATETIME_SUB field will return the subcode for the specific datetime data type (SQL_CODE_DATE, SQL_CODE_TIME or SQL_CODE_TIMESTAMP).

Column 16 SQL_DATETIME_SUB (SMALLINT)
The subtype code for datetime data types. For all other data types this column returns a NULL (including interval data types which DB2 CLI does not support).

Column 17 CHAR_OCTET_LENGTH (INTEGER)
The maximum length in bytes of a character data type column. For all other data types, this column returns a NULL.

Column 18 ORDINAL_POSITION (INTEGER NOT NULL)

Contains the ordinal position of the parameter given by COLUMN_NAME in this result set. This is the ordinal position of the argument to be provided on the CALL statement. The leftmost argument has an ordinal position of 1.

Column 19 IS_NULLABLE (Varchar)

- "NO" if the column does not include NULLs.
- "YES" if the column can include NULLs.
- zero-length string if nullability is unknown.

ISO rules are followed to determine nullability.

An ISO SQL-compliant DBMS cannot return an empty string.

The value returned for this column is different than the value returned for the NULLABLE column. (See the description of the NULLABLE column.)

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLProcedureColumns()` result set in ODBC.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 141. SQLProcedureColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
42601	PARMLIST syntax error.	The PARMLIST value in the stored procedures catalog table contains a syntax error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.

Table 141. SQLProcedureColumns SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY008	Operation canceled	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, SQLCancel() was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, SQLCancel() was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid String or Buffer Length	The value of one of the name length arguments was less than 0, but not equal SQL_NTS.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for procedure name. The connected server does not support *schema* as a qualifier for procedure name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

SQLProcedureColumns() does not return information about the attributes of result sets that may be returned from stored procedures.

If an application is connected to a DB2 server that does not provide support for a stored procedure catalog, or does not provide support for stored procedures, SQLProcedureColumns() will return an empty result set.

Example

```
/* From CLI sample proccols.c */
/* ... */

    printf("Enter Procedure Schema Name Search Pattern:\n");
    gets((char *)proc_schem.s);

    printf("Enter Procedure Name Search Pattern:\n");
    gets((char *)proc_name.s);

    rc = SQLProcedureColumns(hstmt, NULL, 0, proc_schem.s, SQL_NTS,
                             proc_name.s, SQL_NTS, (SQLCHAR *)"%", SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) proc_schem.s, 129,
                    &proc_schem.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) proc_name.s, 129,
                    &proc_name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) column_name.s, 129,
                    &column_name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 5, SQL_C_SHORT, (SQLPOINTER) &arg_type,
                    0, &arg_type_ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) type_name.s, 129,
                    &type_name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 8, SQL_C_LONG, (SQLPOINTER) & length,
                    0, &length_ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 10, SQL_C_SHORT, (SQLPOINTER) &scale,
                    0, &scale_ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 13, SQL_C_CHAR, (SQLPOINTER) remarks.s, 255,
                    &remarks.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        sprintf((char *)cur_name, "%s.%s", proc_schem.s, proc_name.s);
        if (strcmp((char *)cur_name, (char *)pre_name) != 0) {
            printf("\n%s\n", cur_name);
        }
        strcpy((char *)pre_name, (char *)cur_name);
        printf("   %s", column_name.s);
        switch (arg_type)
```

```
        { case SQL_PARAM_INPUT : printf(", Input"); break;
          case SQL_PARAM_OUTPUT : printf(", Output"); break;
          case SQL_PARAM_INPUT_OUTPUT : printf(", Input_Output"); break;
        }
        printf(", %s", type_name.s);
        printf(" (%ld", length);
        if (scale_ind != SQL_NULL_DATA) {
            printf(", %d)\n", scale);
        } else {
            printf(")\n");
        }
        if (remarks.ind > 0 ) {
            printf("(remarks), %s)\n", remarks.s);
        }
    }                           /* endwhile */
```

References

- “SQLProcedures - Get List of Procedure Names” on page 603

SQLProcedures - Get List of Procedure Names

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLProcedures() returns a list of procedure names that have been registered at the server, and which match the specified search pattern.

The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLProcedures    (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLCHAR      FAR  *CatalogName,      /* szProcCatalog */
                SQLSMALLINT       NameLength1,       /* cbProcCatalog */
                SQLCHAR      FAR  *SchemaName,       /* szProcSchema */
                SQLSMALLINT       NameLength2,       /* cbProcSchema */
                SQLCHAR      FAR  *ProcName,         /* szProcName */
                SQLSMALLINT       NameLength3);      /* cbProcName */
```

Function Arguments

Table 142. SQLTables Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.
SQLCHAR *	CatalogName	Input	Catalog qualifier of a 3 part procedure name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	NameLength1	Input	Length of *CatalogName*. This must be set to 0.
SQLCHAR *	SchemaName	Input	Buffer that may contain a *pattern-value* to qualify the result set by schema name. For DB2 for MVS/ESA V 4.1, all the stored procedures are in one schema; the only acceptable value for the *SchemaName* argument is a null pointer. For DB2 Universal Database, *SchemaName* can contain a valid pattern value. For more information about valid search patterns, refer to “Querying System Catalog Information” on page 65.
SQLSMALLINT	NameLength2	Input	Length of *SchemaName*.
SQLCHAR *	ProcName	Input	Buffer that may contain a *pattern-value* to qualify the result set by table name.

Table 142. SQLTables Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	NameLength3	Input	Length of *ProcName*.

Usage

DB2 Universal Database version 5 introduced two system catalog views used to store information about all stored procedures on the server (SYSCAT.PROCEDURES and SYSCAT.PROCPARMS). See "Appendix G. Catalog Views for Stored Procedures" on page 837 for information on these views. `SQLProcedures()` returns a list of stored procedures from these views.

Before version 5, DB2 CLI used the pseudo catalog table for stored procedure registration. By default, DB2 CLI will use the new system catalog views. If the application expects to use the pseudo catalog table then the CLI/ODBC configuration keyword PATCH1 should be set to 262144. See "Replacement of the Pseudo Catalog Table for Stored Procedures" on page 769 for more information.

If the stored procedure is at a DB2 for MVS/ESA V 4.1 server or later, the name of the stored procedures must be registered in the server's SYSIBM.SYSPROCEDURES catalog table.

For other versions of DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set will be returned.

The result set returned by `SQLProcedures()` contains the columns listed in "Columns Returned By SQLProcedures" on page 605 in the order given. The rows are ordered by PROCEDURE_CAT, PROCEDURE_SCHEMA, and PROCEDURE_NAME.

Since calls to `SQLProcedures()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

If the SQL_ATTR_LONGDATA_COMPAT connection attribute is set, LOB column types will be reported as LONG VARCHAR, LONG VARBINARY, or LONG VARGRAPHIC types.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns Returned By SQLProcedures

Column 1 PROCEDURE_CAT (VARCHAR(128))
This is always null.

Column 2 PROCEDURE_SCHEM (VARCHAR(128))
The name of the schema containing PROCEDURE_NAME.

Column 3 PROCEDURE_NAME (VARCHAR(128) NOT NULL)
The name of the procedure.

Column 4 NUM_INPUT_PARAMS (INTEGER not NULL)
Number of input parameters.

This column should not be used, it is reserved for future use by ODBC.

It was used in versions of DB2 CLI before version 5. For backward compatibility it can be used with the old DB2CLI.PROCEDURES pseudo catalog table (by setting the PATCH1 CLI/ODBC Configuration keyword). See “Replacement of the Pseudo Catalog Table for Stored Procedures” on page 769 for more information.

Column 5 NUM_OUTPUT_PARAMS (INTEGER not NULL)
Number of output parameters.

This column should not be used, it is reserved for future use by ODBC.

It was used in versions of DB2 CLI before version 5. For backward compatibility it can be used with the old DB2CLI.PROCEDURES pseudo catalog table (by setting the PATCH1 CLI/ODBC Configuration keyword). See “Replacement of the Pseudo Catalog Table for Stored Procedures” on page 769 for more information.

Column 6 NUM_RESULT_SETS (INTEGER not NULL)
Number of result sets returned by the procedure.

This column should not be used, it is reserved for future use by ODBC.

It was used in versions of DB2 CLI before version 5. For backward compatibility it can be used with the old DB2CLI.PROCEDURES pseudo catalog table (by setting the PATCH1 CLI/ODBC

Configuration keyword). See “Replacement of the Pseudo Catalog Table for Stored Procedures” on page 769 for more information.

Column 7 REMARKS (VARCHAR(254))
Contains the descriptive information about the procedure.

Column 8 PROCEDURE_TYPE (SMALLINT)
Defines the procedure type:

- SQL_PT_UNKNOWN: It cannot be determined whether the procedure returns a value.
- SQL_PT_PROCEDURE: The returned object is a procedure; that is, it does not have a return value.
- SQL_PT_FUNCTION: The returned object is a function; that is, it has a return value.

DB2 CLI always returns SQL_PT_PROCEDURE.

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLProcedures()` result set in ODBC.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 143. SQLProcedures SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.

Table 143. SQLProcedures SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, SQLCancel() was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, SQLCancel() was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for procedure name. The connected server does not supported schema as a qualifier for procedure name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

If an application is connected to a DB2 server that does not provide support for a stored procedure catalog, or does not provide support for stored procedures, SQLProcedureColumns() will return an empty result set.

Example

```
/* From CLI sample procs.c */
/* ... */

    printf("Enter Procedure Schema Name Search Pattern:\n");
    gets((char *)proc_schem.s);

    rc = SQLProcedures(hstmt, NULL, 0, proc_schem.s, SQL_NTS,
                    (SQLCHAR *)"%", SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) proc_schem.s, 129,
                    &proc_schem.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) proc_name.s, 129,
                    &proc_name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) remarks.s, 255,
                    &remarks.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf("PROCEDURE SCHEMA          PROCEDURE NAME            \n");
    printf("------------------------- ------------------------- \n");
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("%-25s %-25s\n", proc_schem.s, proc_name.s);
        if (remarks.ind != SQL_NULL_DATA) {
            printf("  (Remarks) %s\n", remarks.s);
        }
    }                           /* endwhile */
```

References

- “SQLProcedureColumns - Get Input/Output Parameter Information for A Procedure” on page 594

SQLPutData - Passing Data Value for A Parameter

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	ISO CLI

SQLPutData() is called following an SQLParamData() call returning SQL_NEED_DATA to supply parameter data values. This function can be used to send large parameter values in pieces.

The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLPutData      (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLPOINTER        DataPtr,           /* rgbValue */
                SQLINTEGER        StrLen_or_Ind);    /* cbValue */
```

Function Arguments

Table 144. SQLPutData Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.
SQLPOINTER	DataPtr	Input	Pointer to the actual data, or portion of data, for a parameter. The data must be in the form specified in the SQLBindParameter() call that the application used when specifying the parameter.
SQLINTEGER	StrLen_or_Ind	Input	The length of *DataPtr*. Specifies the amount of data sent in a call to SQLPutData() . The amount of data can vary with each call for a given parameter. The application can also specify SQL_NTS or SQL_NULL_DATA for *StrLen_or_Ind*. *StrLen_or_Ind* is ignored for all fixed length C buffer types, such as date, time, timestamp, and all numeric C buffer types. For cases where the C buffer type is SQL_C_CHAR or SQL_C_BINARY, or if SQL_C_DEFAULT is specified as the C buffer type and the C buffer type default is SQL_C_CHAR or SQL_C_BINARY, this is the number of bytes of data in the *DataPtr* buffer.

Usage

For a description on the SQLParamData() and SQLPutData() sequence, refer to "Sending/Retrieving Long Data in Pieces" on page 80.

The application calls SQLPutData() after calling SQLParamData() on a statement in the SQL_NEED_DATA state to supply the data values for an SQL_DATA_AT_EXEC parameter. Long data can be sent in pieces via repeated calls to SQLPutData(). After all the pieces of data for the parameter have been sent, the application calls SQLParamData() again to proceed to the next SQL_DATA_AT_EXEC parameter, or, if all parameters have data values, to execute the statement.

SQLPutData() cannot be called more than once for a fixed length C buffer type, such as SQL_C_LONG.

After an SQLPutData() call, the only legal function calls are SQLParamData(), SQLCancel(), or another SQLPutData() if the input data is character or binary data. As with SQLParamData(), all other function calls using this statement handle will fail. In addition, all function calls referencing the parent *hdbc* of *StatementHandle* will fail if they involve changing any attribute or state of that connection; that is, the following function calls on the parent *hdbc* are also not permitted:

- SQLAllocConnect()
- SQLAllocStmt()
- SQLSetConnectAttr()
- SQLNativeSql()
- SQLTransact()

Should they be invoked during an SQL_NEED_DATA sequence, these function will return SQL_ERROR with SQLSTATE of **HY**010 and the processing of the SQL_DATA_AT_EXEC parameters will not be affected.

If one or more calls to SQLPutData() for a single parameter results in SQL_SUCCESS, attempting to call SQLPutData() with *StrLen_or_Ind* set to SQL_NULL_DATA for the same parameter results in an error with SQLSTATE of **22**005. This error does not result in a change of state; the statement handle is still in a *Need Data* state and the application can continue sending parameter data.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING

- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Some of the following diagnostics conditions may also be reported on the final SQLParamData() call rather than at the time the SQLPutData() is called.

Table 145. SQLPutData SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data sent for a numeric parameter was truncated without the loss of significant digits. Timestamp data sent for a date or time column was truncated. Function returns with SQL_SUCCESS_WITH_INFO.
22001	String data right truncation.	More data was sent for a binary or char data than the data source can support for that column.
22003	Numeric value out of range.	The data sent for a numeric parameter cause the whole part of the number to be truncated when assigned to the associated column. SQLPutData() was called more than once for a fixed length parameter.
22005	Error in assignment.	The data sent for a parameter was incompatible with the data type of the associated table column.
22007	Invalid datetime format.	The data value sent for a date, time, or timestamp parameters was invalid.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, SQLCancel() was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, SQLCancel() was called on the *StatementHandle* from a different thread in a multithread application.

Table 145. SQLPutData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY009	Invalid argument value.	The argument *DataPtr* was a NULL pointer, and the argument *StrLen_or_Ind* was neither 0 nor SQL_NULL_DATA.
HY010	Function sequence error.	The statement handle *StatementHandle* must be in a need data state and must have been positioned on an SQL_DATA_AT_EXEC parameter via a previous SQLParamData() call.
HY090	Invalid string or buffer length.	The argument *DataPtr* was not a NULL pointer, and the argument *StrLen_or_Ind* was less than 0, but not equal to SQL_NTS or SQL_NULL_DATA.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

A new value for *StrLen_or_Ind*, SQL_DEFAULT_PARAM, was introduced in ODBC 2.0, to indicate that the procedure is to use the default value of a parameter, rather than a value sent from the application. Since DB2 stored procedure arguments do not have the concept of default values, specification of this value for *StrLen_or_Ind* argument will result in an error when the CALL statement is executed since the SQL_DEFAULT_PARAM value will be considered an invalid length.

ODBC 2.0 also introduced the SQL_LEN_DATA_AT_EXEC(*length*) macro to be used with the *StrLen_or_Ind* argument. The macro is used to specify the sum total length of the entire data that would be sent for character or binary C data via the subsequent SQLPutData() calls. Since the DB2 ODBC driver does not need this information, the macro is not needed. An ODBC application calls SQLGetInfo() with the SQL_NEED_LONG_DATA_LEN option to check if the driver needs this information. The DB2 ODBC driver will return 'N' to indicate that this information is not needed by SQLPutData().

Example

```
/* From CLI sample picin2.c */
/* ... */
    SQLCHAR * stmt =
    "INSERT INTO emp_photo (empno, photo_format, picture) VALUES (?, ?, ?)" ;
/* ... */
    /* Prepare the statement */
    rc = SQLPrepare(hstmt, stmt, SQL_NTS);
```

```
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          6, 0, Empno, 7, NULL);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          10, 0,  Photo_Format, 11, NULL);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /*
     * This paramter will use SQLPutData, rgbValue is set to Param Number,
     * pcbValue is set to SQL_DATA_AT_EXEC
     */
    PicLength = SQL_DATA_AT_EXEC;
    rc = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_BLOB,
                          BUFSIZ, 0, (SQLPOINTER)input_param,
                          BUFSIZ, &PicLength);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
/* ... */
    if ( ( rc = SQLExecute( hstmt ) ) == SQL_NEED_DATA ) {
       PicFile = fopen((char *)FName, "rb");
       if (PicFile == NULL) {
          printf(">---- ERROR Opening File -------");
          /* Cancel the DATA AT EXEC state for hstmt */
          rc = SQLCancel(hstmt);
          CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
          /* Commit, free resources, disconnect and exit */
       }
       else {
          while ( ( rc = SQLParamData( hstmt, ( SQLPOINTER ) &prgbValue )
                  ) == SQL_NEED_DATA
                ) {
             printf("Getting data for %s\n", prgbValue);
             /*
              if more than 1 parms used DATA_AT_EXEC then prgbValue would
              have to be checked to determine which param needed data
             */
             while ( feof( PicFile ) == 0 ) {
                n = fread(fbuffer, sizeof(char), BUFSIZ, PicFile);
                rc = SQLPutData(hstmt, fbuffer, n);
                CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
                FileSize = FileSize + n;
                if (FileSize > 102400u) {
                   /* BLOB column defined as 100K MAX */
                   printf(">---- ERROR: File > 100K  -------");
                   exit(terminate(hdbc, SQL_ERROR));
                }
             }
             printf("Read a total of %u bytes from %s\n", FileSize, FName);
          }
          CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

```
    }
  }
  else CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

References

- "SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator" on page 247
- "SQLExecute - Execute a Statement" on page 373
- "SQLExecDirect - Execute a Statement Directly" on page 365
- "SQLParamData - Get Next Parameter For Which A Data Value Is Needed" on page 575
- "SQLCancel - Cancel Statement" on page 290

SQLRowCount - Get Row Count

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLRowCount() returns the number of rows in a table that were affected by an UPDATE, INSERT, or DELETE statement executed against the table, or a view based on the table.

SQLExecute() or SQLExecDirect() must be called before calling this function.

Syntax

```
SQLRETURN   SQLRowCount      (SQLHSTMT          StatementHandle,   /* hstmt */
                              SQLINTEGER   FAR  *RowCountPtr);     /* pcrow */
```

Function Arguments

Table 146. SQLRowCount Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLINTEGER *	*RowCountPtr*	output	Pointer to location where the number of rows affected is stored.

Usage

If the last executed statement referenced by the input statement handle was not an UPDATE, INSERT, or DELETE statement, or if it did not execute successfully, then the function sets the contents of *RowCountPtr* to -1.

Any rows in other tables that may have been affected by the statement (for example, cascading deletes) are not included in the count.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 147. SQLRowCount SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.

Table 147. SQLRowCount SQLSTATEs (continued)

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function was called prior to calling `SQLExecute()` or `SQLExecDirect()` for the *StatementHandle*.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Authorization

None.

Example

Refer to "Example" on page 340.

References

- "SQLExecDirect - Execute a Statement Directly" on page 365
- "SQLExecute - Execute a Statement" on page 373
- "SQLNumResultCols - Get Number of Result Columns" on page 572

SQLSetColAttributes - Set Column Attributes

Deprecated

Note:

In ODBC version 3, `SQLSetColAttributes()` has been deprecated.

Although this version of DB2 CLI continues to support `SQLSetColAttributes()`, we recommend that you stop using it in your DB2 CLI programs so that they conform to the latest standards. All arguments you pass to `SQLSetColAttributes()` will be ignored, and the function call will always return SQL_SUCCESS.

Now that DB2 CLI uses deferred prepare by default, there is no need for the functionality of `SQLSetColAttributes()`. See "Deferred Prepare now on by Default" on page 773 for more details.

SQLSetConnectAttr - Set Connection Attributes

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLSetConnectAttr() sets attributes that govern aspects of connections.

Syntax

```
SQLRETURN    SQLSetConnectAttr(SQLHDBC           ConnectionHandle,
                               SQLINTEGER        Attribute,
                               SQLPOINTER        ValuePtr,
                               SQLINTEGER        StringLength);
```

Function Arguments

Table 148. SQLSetConnectAttr Arguments

Data Type	Argument	Use	Description
SQLHDBC	ConnectionHandle	input	Connection handle.
SQLINTEGER	*Attribute*	input	Attribute to set, listed in "Attribute Values" on page 621.
SQLPOINTER	ValuePtr	input	Pointer to the value to be associated with *Attribute*. Depending on the value of *Attribute*, **ValuePtr* will be a 32-bit unsigned integer value or point to a null-terminated character string. Note that if the *Attribute* argument is a driver-specific value, the value in **ValuePtr* may be a signed integer.

Table 148. SQLSetConnectAttr Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER	*StringLength*	input	If *Attribute* is an ODBC-defined attribute and *ValuePtr* points to a character string or a binary buffer, this argument should be the length of **ValuePtr*. If *Attribute* is an ODBC-defined attribute and *ValuePtr* is an integer, *StringLength* is ignored. If *Attribute* is a DB2 CLI attribute, the application indicates the nature of the attribute by setting the *StringLength* argument. *StringLength* can have the following values: • If *ValuePtr* is a pointer to a character string, then *StringLength* is the length of the string or SQL_NTS. • If *ValuePtr* is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in *StringLength*. This places a negative value in *StringLength*. • If *ValuePtr* is a pointer to a value other than a character string or a binary string, then *StringLength* should have the value SQL_IS_POINTER. • If *ValuePtr* contains a fixed-length value, then *StringLength* is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.

Usage

Setting Statement Attributes using SQLSetConnectAttr() no Longer Supported

The ability to set statement attributes using `SQLSetConnectAttr()` is no longer supported. To support applications written before version 5, some statement attributes can be set using `SQLSetConnectAttr()` in this release of DB2 CLI. All applications that rely on this behavior, however, should be updated to use `SQLSetStmtAttr()` instead. See "Setting a Subset of Statement Attributes using SQLSetConnectAttr()" on page 770 for more information.

For version 2 applications that continue to set statement attributes using `SQLSetConnectAttr()`, if an error is returned when a statement attribute is set on one of multiple active statements, the statement attribute is established as the default for statements later allocated on the connection, but it is undefined whether statement attributes previously set on the same call to

SQLSetConnectAttr() remain set after the function is aborted and the error is returned. This is one reason why SQLSetConnectAttr() should not be used to set statement attributes.

If SQLSetConnectAttr() is called to set a statement attribute that sets the header field of a descriptor, the descriptor field is set for the application descriptors currently associated with all statements on the connection. However, the attribute setting does not affect any descriptors that may be associated with the statements on that connection in the future.

Connection Attributes

The currently defined attributes and the version of DB2 CLI in which they were introduced are shown below; it is expected that more will be defined to take advantage of different data sources.

An application can call SQLSetConnectAttr() at any time between the time the connection is allocated or freed. All connection and statement attributes successfully set by the application for the connection persist until SQLFreeHandle() is called on the connection.

Some connection attributes can be set only before a connection has been made; others can be set only after a connection has been made, while some cannot be set once a statement is allocated. The following table indicates when each of the connection attributes can be set.

Table 149. When Connection Attributes can be Set

Attribute	Before connection	After connection	After statements allocated
SQL_ATTR_ACCESS_MODE	Yes	Yes	Yes [a]
SQL_ATTR_ASYNC_ENABLE	Yes	Yes	No [b]
SQL_ATTR_AUTO_IPD (read only)	No	Yes	Yes
SQL_ATTR_AUTOCOMMIT	Yes	Yes	Yes [c]
SQL_ATTR_CLISCHEMA	Yes	Yes	Yes
SQL_ATTR_CLOSE_BEHAVIOR	No	Yes	Yes [c]
SQL_ATTR_CONN_CONTEXT	Yes	No	No
SQL_ATTR_CONNECT_NODE	Yes	No	No
SQL_ATTR_CONNECTTYPE	Yes	No	No
SQL_ATTR_CURRENT_SCHEMA	Yes	Yes	Yes
SQL_ATTR_DB2_SQLERRP (read only)	No	Yes	Yes
SQL_ATTR_DB2ESTIMATE	No	Yes	Yes
SQL_ATTR_DB2EXPLAIN	No	Yes	Yes
SQL_ATTR_ENLIST_IN_DTC	No	Yes	Yes
SQL_ATTR_INFO_ACCTSTR	No	Yes	Yes
SQL_ATTR_INFO_APPLNAME	No	Yes	Yes

Table 149. When Connection Attributes can be Set (continued)

Attribute	Before connection	After connection	After statements allocated
SQL_ATTR_INFO_USERID	No	Yes	Yes
SQL_ATTR_INFO_WRKSTNNAME	No	Yes	Yes
SQL_ATTR_LOGIN_TIMEOUT	Yes	No	No
SQL_ATTR_LONGDATA_COMPAT	Yes	Yes	Yes
SQL_ATTR_MAXCONN	Yes	No	No
SQL_ATTR_OPTIMIZESQLCOLUMNS	Yes	Yes	Yes
SQL_ATTR_QUIET_MODE	Yes	Yes	Yes
SQL_ATTR_SYNC_POINT	Yes	No	No
SQL_ATTR_TRANSLATE_OPTION	Yes	Yes	Yes
SQL_ATTR_TXN_ISOLATION	No	Yes [c]	Yes [a]
SQL_ATTR_WCHARTYPE	Yes	Yes [c]	Yes [c]

[a] Will only affect subsequently allocated statements.

[b] Attribute must be set before there is an active statement.

[c] Attribute can be set only if there are no open transactions on the connection.

Some connection attributes support substitution of a similar value if the data source does not support the value specified in **ValuePtr*. In such cases, DB2 CLI returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed.). For example, if the application attempts to set SQL_ATTR_ASYNC_ENABLE to SQL_ASYNC_ENABLE_ON and the server does not support this then DB2 CLI substitutes the value SQL_ASYNC_ENABLE_OFF instead. To determine the substituted value, an application calls `SQLGetConnectAttr()`.

The format of information set through **ValuePtr* depends on the specified *Attribute*. `SQLSetConnectAttr()` will accept attribute information in one of two different formats: a null-terminated character string or a 32-bit integer value. The format of each is noted in the attribute's description. Character strings pointed to by the *ValuePtr* argument of `SQLSetConnectAttr()` have a length of *StringLength* bytes.

The *StringLength* argument is ignored if the length is defined by the attribute, as is the case for all attributes introduced before DB2 CLI Version 5.2.

Attribute Values

The following version 2 connection attributes (that were set using `SQLSetConnectOption()`) have been renamed for version 5:

Table 150. Renamed Connection Attributes

Version 2 name	Version 5 and later name
SQL_ACCESS_MODE	SQL_ATTR_ACCESS_MODE
SQL_AUTOCOMMIT	SQL_ATTR_AUTOCOMMIT
SQL_CONNECTTYPE	SQL_ATTR_CONNECTTYPE
SQL_CURRENT_SCHEMA	SQL_ATTR_CURRENT_SCHEMA
SQL_DB2ESTIMATE	SQL_ATTR_DB2ESTIMATE
SQL_DB2EXPLAIN	SQL_ATTR_DB2EXPLAIN
SQL_LOGIN_TIMEOUT	SQL_ATTR_LOGIN_TIMEOUT
SQL_LONGDATA_COMPAT	SQL_ATTR_LONGDATA_COMPAT
SQL_MAXCONN	SQL_ATTR_MAXCONN
SQL_QUIET_MODE	SQL_ATTR_QUIET_MODE
SQL_SCHEMA	SQL_ATTR_SCHEMA
SQL_SYNC_POINT	SQL_ATTR_SYNC_POINT
SQL_TXN_ISOLATION	SQL_ATTR_TXN_ISOLATION
SQL_WCHARTYPE	SQL_ATTR_WCHARTYPE

Attribute

***ValuePtr Contents**

SQL_ATTR_ACCESS_MODE (DB2 CLI v2)

A 32-bit integer value which can be either:

- SQL_MODE_READ_ONLY: the application is indicating that it will not be performing any updates on data from this point on. Therefore, a less restrictive isolation level and locking can be used on transactions; that is uncommitted read (SQL_TXN_READ_UNCOMMITTED).

  DB2 CLI does not ensure that requests to the database are *read-only*. If an update request is issued, DB2 CLI will process it using the transaction isolation level it has selected as a result of the SQL_MODE_READ_ONLY setting.
- **SQL_MODE_READ_WRITE**: the application is indicating that it will be making updates on data from this point on. DB2 CLI will go back to using the default transaction isolation level for this connection.

  SQL_MODE_READ_WRITE is the default.

There must not be any outstanding transactions on this connection.

SQL_ATTR_ASYNC_ENABLE (DB2 CLI v5)

A 32-bit integer value that specifies whether a function called with a statement on the specified connection is executed asynchronously:

- **SQL_ASYNC_ENABLE_OFF** = Off (the default)
- SQL_ASYNC_ENABLE_ON = On

Setting SQL_ASYNC_ENABLE_ON enables asynchronous execution for all future statement handles allocated on this connection. This also enables asynchronous execution for existing statement handles associated with this connection. An error is returned if asynchronous execution is turned on while there is an active statement on the connection.

This attribute can be set whether SQLGetInfo(), called with the *InfoType* SQL_ASYNC_MODE, returns SQL_AM_CONNECTION or SQL_AM_STATEMENT.

Once a function has been called asynchronously, only the original function, SQLAllocHandle(), SQLCancel(), SQLGetDiagField(), or SQLGetDiagRec() can be called on the statement or the connection associated with *StatementHandle*, until the original function returns a code other than SQL_STILL_EXECUTING. Any other function called on *StatementHandle* or the connection associated with *StatementHandle* returns SQL_ERROR with an SQLSTATE of HY010 (Function sequence error). Functions can be called on other statements.

In general, applications should execute functions asynchronously only on single-threaded operating systems. On multi-threaded operating systems, applications should execute functions on separate threads, rather than executing them asynchronously on the same thread. Applications that only operate on multi-threaded operating systems do not need to support asynchronous execution. For more information, see “Writing Multi-Threaded Applications” on page 46 and “Asynchronous Execution of CLI” on page 138.

The following functions can be executed asynchronously:

```
SQLColAttribute()          SQLGetTypeInfo()
SQLColumnPrivileges()      SQLMoreResults()
SQLColumns()               SQLNumParams()
SQLCopyDesc()              SQLNumResultCols()
SQLDescribeCol()           SQLParamData()
SQLDescribeParam()         SQLPrepare()
SQLExecDirect()            SQLPrimaryKeys()
SQLExecute()               SQLProcedureColumns()
SQLFetch()                 SQLProcedures()
SQLFetchScroll()           SQLPutData()
SQLForeignKeys()           SQLSetPos()
SQLGetData()               SQLSpecialColumns()
SQLGetDescField() 1*       SQLStatistics()
SQLGetDescRec() 1*         SQLTablePrivileges()
SQLGetDiagField()          SQLTables()
SQLGetDiagRec()
```

```
1* These functions can be called asynchronously only if the descriptor
   is an implementation descriptor, not an application descriptor.
```

SQL_ATTR_AUTO_IPD (DB2 CLI v5)

A read-only 32-bit integer value that specifies whether automatic population of the IPD after a call to SQLPrepare() is supported:

- SQL_TRUE = Automatic population of the IPD after a call to SQLPrepare() is supported by the server.
- SQL_FALSE = Automatic population of the IPD after a call to SQLPrepare() is not supported by the server. Servers that do not support prepared statements will not be able to populate the IPD automatically.

If SQL_TRUE is returned for the SQL_ATTR_AUTO_IPD connection attribute, the statement attribute SQL_ATTR_ENABLE_AUTO_IPD can be set to turn automatic population of the IPD on or off. If SQL_ATTR_AUTO_IPD is SQL_FALSE, SQL_ATTR_ENABLE_AUTO_IPD cannot be set to SQL_TRUE.

The default value of SQL_ATTR_ENABLE_AUTO_IPD is equal to the value of SQL_ATTR_AUTO_IPD.

This connection attribute can be returned by SQLGetConnectAttr(), but cannot be set by SQLSetConnectAttr().

SQL_ATTR_AUTOCOMMIT (DB2 CLI v2)

A 32-bit integer value that specifies whether to use auto-commit or manual commit mode:

- SQL_AUTOCOMMIT_OFF: the application must manually, explicitly commit or rollback transactions with SQLTransact() calls.
- **SQL_AUTOCOMMIT_ON**: DB2 CLI operates in auto-commit mode. Each statement is implicitly committed. Each statement, that is not a query, is committed immediately after it has been executed. Each query is committed immediately after the associated cursor is closed.

  SQL_AUTOCOMMIT_ON is the default.

  Note: If this is a coordinated distributed unit of work connection, then the default is **SQL_AUTOCOMMIT_OFF**

Since in many DB2 environments, the execution of the SQL statements and the commit may be flowed separately to the database server, autocommit can be expensive. It is recommended that the application developer take this into consideration when selecting the auto-commit mode.

Note: Changing from manual commit to auto-commit mode will commit any open transaction on the connection.

DB2 CLI Version 1 applications assume the default is manual commit mode. Refer to "Incompatibilities" on page 768.

SQL_ATTR_CLISCHEMA (DB2 CLI v6)

A null-terminated character string containing the name of the DB2 ODBC catalog view stored on the host DBMS to use.

The DB2 ODBC catalog is designed to improve the performance of schema calls for lists of tables in ODBC applications that connect to host DBMSs through DB2 Connect.

The DB2 ODBC catalog, created and maintained on the host DBMS, contains rows representing objects defined in the real DB2 catalog, but these rows include only the columns necessary to support ODBC operations. The tables in the DB2 ODBC catalog are pre-joined and specifically indexed to support fast catalog access for ODBC applications.

System administrators can create multiple DB2 ODBC catalog views, each containing only the rows that are needed by a particular user group. Each end user can then select the DB2 ODBC catalog view they wish to use (by setting this attribute).

While this attribute has some similar effects as the SYSSCHEMA keyword, SQL_ATTR_CLISCHEMA should be used instead (where applicable). SQL_ATTR_CLISCHEMA improves data access efficiency: The user-defined tables used with SYSSCHEMA were mirror images of the DB2 catalog tables, and the ODBC driver still had to join rows from multiple tables to produce the information required by the ODBC user. Using SQL_ATTR_CLISCHEMA also results in less contention on the catalog tables.

There is also a corresponding DB2 CLI/ODBC Driver configuration keyword; "CLISCHEMA" on page 169

SQL_ATTR_CLOSE_BEHAVIOR (DB2 CLI v6)

A 32-bit integer that specifies whether the DB2 server should attempt to release read locks acquired during a cursor's operation when the cursor is closed. It can be set to either:

- **SQL_CC_NO_RELEASE** - read locks are not released. This is the default.
- SQL_CC_RELEASE - read locks are released.

For cursors opened with isolation UR or CS, read locks are not held after a cursor moves off a row. For cursors opened with isolation RS

or RR, SQL_ATTR_CLOSE_BEHAVIOR modifies some of those isolation levels, and an RR cursor may experience nonrepeatable reads or phantom reads.

If a cursor that is originally RR or RS is reopened after being closed with SQL_ATTR_CLOSE_BEHAVIOR then new read locks will be acquired.

For more information see "SQLCloseCursor - Close Cursor and Discard Pending Results" on page 293.

SQL_ATTR_CONN_CONTEXT (DB2 CLI v5)

Indicates which context the connection should use. An SQLPOINTER to either:

- a valid context (allocated by the `sqleBeginCtx()` DB2 API) to set the context
- a NULL pointer to reset the context

This attribute can only be used when the application is using the DB2 context APIs to manage multi-threaded applications. By default, DB2 CLI manages contexts by allocating one context per connection handle, and ensuring that any executing thread is attached to the correct context.

For more information about when an application may have to manage contexts, refer to "Writing Multi-Threaded Applications" on page 46.

For more information about contexts, refer to the `sqleBeginCtx()` API in the *Administrative API Reference*.

SQL_ATTR_CONNECT_NODE (DB2 CLI v6)

This attribute is used to specify the target logical node of a DB2 Extended Enterprise Edition database partition server that you want to connect to. This setting overrides the value of the environment variable DB2NODE. It can be set to:

- an integer between 0 and 999
- SQL_CONN_CATALOG_NODE

If this variable is not set, the target logical node defaults to the logical node which is defined with port 0 on the machine.

There is also a corresponding DB2 CLI/ODBC Driver configuration keyword; "CONNECTTYPE" on page 170.

SQL_ATTR_CONNECTION_DEAD (DB2 CLI v6)

A READ ONLY 32-bit integer value that indicates whether or not the connection is still active. DB2 CLI will return one of the following values:

- SQL_CD_FALSE - the conneciton is still active.
- SQL_CD_TRUE - an error has already happend and caused the connection to the server to be disconnected. The user should still perform a disconnect and to cleanup the DB2CLI resources.

This attribute is used mainly by the Microsoft ODBC Driver Manager 3.5x before pooling the connection.

SQL_ATTR_CONNECTION_TIMEOUT (DB2 CLI v5)

This connection attribute is defined by ODBC, but is not supported by DB2 CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

A 32-bit integer value corresponding to the number of seconds to wait for any request on the connection to complete before returning to the application.

DB2 CLI always behaves as if *ValuePtr* was set to 0 (the default); there is no time out.

SQL_ATTR_CONNECTTYPE (DB2 CLI v2)

A 32-bit integer value that specifies whether this application is to operate in a coordinated or uncoordinated distributed environment. If the processing needs to be coordinated, then this option must be considered in conjunction with the SQL_ATTR_SYNC_POINT connection option. The possible values are:

- **SQL_CONCURRENT_TRANS**: The application can have concurrent multiple connections to any one database or to multiple databases. Each connection has its own commit scope. No effort is made to enforce coordination of transaction. If an application issues a commit using the environment handle on `SQLTransact()` and not all of the connections commit successfully, the application is responsible for recovery.

  The current setting of the SQL_ATTR_SYNC_POINT option is ignored.

  This is the default.
- SQL_COORDINATED_TRANS: The application wishes to have commit and rollbacks coordinated among multiple database connections. This option setting corresponds to the specification of the Type 2 CONNECT in embedded SQL and must be considered in conjunction with the SQL_ATTR_SYNC_POINT connection

option. In contrast to the SQL_CONCURRENT_TRANS setting described above, the application is permitted only one open connection per database.

Note: This connection type results in the default for SQL_ATTR_AUTOCOMMIT connection option to be SQL_AUTOCOMMIT_OFF.

This option must be set before making a connect request; otherwise, the `SQLSetConnectOption()` call will be rejected.

All the connections within an application must have the same SQL_ATTR_CONNECTTYPE and SQL_ATTR_SYNC_POINT values. The first connection determines the acceptable attributes for the subsequent connections. We recommend that the application set the SQL_ATTR_CONNECTTYPE attribute at the environment level rather than on a per connection basis. ODBC applications written to take advantage of coordinated DB2 transactions must set these attributes at the connection level for each connection as `SQLSetEnvAttr()` is not supported in ODBC.

The default connect type can also be set using the CONNECTTYPE DB2 CLI/ODBC configuration keyword. See "Configuring db2cli.ini" on page 159 for more information.

Note: This is an IBM defined extension.

SQL_ATTR_CURRENT_CATALOG (DB2 CLI v5)

This connection attribute is defined by ODBC, but is not supported by DB2 CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

A null-terminated character string containing the name of the catalog to be used by the data source.

SQL_ATTR_CURRENT_SCHEMA (DB2 CLI v2)

A null-terminated character string containing the name of the schema to be used by DB2 CLI for the `SQLColumns()` call if the *szSchemaName* pointer is set to null.

To reset this option, specify this option with a zero length or a null pointer for the *ValuePtr* argument.

This option is useful when the application developer has coded a generic call to `SQLColumns()` that does not restrict the result set by schema name, but needs to constrain the result set at isolated places in the code.

This option can be set at any time and will be effective on the next SQLColumns() call where the *szSchemaName* pointer is null.

Note: This is an IBM defined extension.

SQL_ATTR_DB2_SQLERRP (DB2 CLI v6)

A read-only, null-terminated string containing the *sqlerrp* field of the sqlca.

Begins with a three-letter identifier indicating the product, followed by five digits indicating the version, release, and modification level of the product. For example, SQL05000 means DB2 Universal Database versions for Version 5 Release 0 Modification level 0.

If SQLCODE indicates an error condition, then this field identifies the module that returned the error.

This field is also used when a successful connection is completed.

See the *SQL Reference* for more information on the sqlca and this particular field.

SQL_ATTR_DB2ESTIMATE (DB2 CLI v2)

A 32-bit integer that specified whether DB2 CLI will display a dialog window to report estimates returned by the optimizer at the end of SQL query preparation.

- **0** : Estimates are not returned.

  This is the default.
- very large positive integer: The threshold above which DB2 CLI will pop up a window to report estimates. This positive integer value is compared against the SQLERRD(4) field in the SQLCA associated with the PREPARE. If the DB2ESTIMATE value is greater, the estimates window will appear.

  The graphical window will display optimizer estimates, along with push buttons to allow the user to choose whether they wish to continue with subsequent execution of this query or to cancel it.

  The recommended value for this option is 60000.

This option is used in conjunction with SQL_ATTR_QUIET_MODE and is applicable only to applications with graphical user interfaces. The application can implement this feature directly without using this option by calling SQLGetSQLCA() after an SQLPrepare() for a query and then displaying the appropriate information, thus allowing a more integrated overall interface.

The new SQL_ATTR_DB2ESTIMATE setting is effective on the next statement preparation for this connection.

Note: This is an IBM defined extension.

SQL_ATTR_DB2EXPLAIN (DB2 CLI v2)

A 32-bit integer that specifies whether Explain snapshot and/or Explain mode information should be generated by the server:

- SQL_DB2EXPLAIN_OFF: Both the Explain Snapshot and the Explain table option facilities are disabled (a SET CURRENT EXPLAIN SNAPSHOT=NO and a SET CURRENT EXPLAIN MODE=NO are sent to the server).
- SQL_DB2EXPLAIN_SNAPSHOT_ON: The Explain Snapshot facility is enabled, and the Explain table option facility is disabled (a SET CURRENT EXPLAIN SNAPSHOT=YES and a SET CURRENT EXPLAIN MODE=NO are sent to the server).
- SQL_DB2EXPLAIN_MODE_ON: The Explain Snapshot facility is disabled, and the Explain table option facility is enabled (a SET CURRENT EXPLAIN SNAPSHOT=NO and a SET CURRENT EXPLAIN MODE=YES are sent to the server).
- SQL_DB2EXPLAIN_SNAPSHOT_MODE_ON: Both the Explain Snapshot and the Explain table option facilities are enabled (a SET CURRENT EXPLAIN SNAPSHOT=YES and a SET CURRENT EXPLAIN MODE=YES are sent to the server).

Before the explain information can be generated, the explain tables must be created. See the *SQL Reference* for additional information.

This statement is not under transaction control and is not affected by a ROLLBACK. The new SQL_ATTR_DB2EXPLAIN setting is effective on the next statement preparation for this connection.

The current authorization ID must have INSERT privilege for the Explain tables.

This value can also be set using the DB2EXPLAIN DB2 CLI/ODBC configuration keyword. See “Configuring db2cli.ini” on page 159 for more information.

Note: This is an IBM defined extension.

SQL_ATTR_ENLIST_IN_DTC (DB2 CLI v5.2)

An SQLPOINTER which can be either:

- non-null transaction pointer:

  The application is asking the DB2 CLI/ODBC driver to change the state of the connection from non-distributed transaction state to distributed state. The connection will be enlisted with the Distributed Transaction Coordinator (DTC).

- null:

  The application is asking the DB2 CLI/ODBC driver to change the state of the connection from distributed transaction state to a non-distributed transaction state.

This attribute is only used in a Microsoft Transaction Server (MTS) environment to enlist or un-enlist a connection with MTS.

Each time this attribute is used with a non-null transaction pointer, the previous transaction is assumed to be ended and a new transaction is initiated. The application must call the ITransaction member function Endtransaction before calling this API with a non-null pointer. Otherwise the previous transaction will be aborted. The application can enlist mutiple connections with the same transaction pointer.

Note: This connection attribute is specified by MTS automatically for each transaction and is not coded by the user application. It is imperative for CLI/ODBC applications that there will be no concurrent SQL statements executing on 2 different connections into the same database that are enlisted in the same transaction.

SQL_ATTR_INFO_ACCTSTR (DB2 CLI v6)

A null-terminated character string used to identify the client accounting string sent to the host database server when using DB2 Connect. Please note:

- When the value is being set, some servers may not handle the entire length provided and may truncate the value.
- DB2 for OS/390 servers support up to a length of 200 characters.
- To ensure that the data is converted correctly when transmitted to a DRDA server, use only the characters A to Z, 0 to 9, and the underscore(_) or period (.).

Note: This is an IBM defined extension.

SQL_ATTR_INFO_APPLNAME (DB2 CLI v6)

A null-terminated character string used to identify the client application name sent to the host database server when using DB2 Connect. Please note:

- When the value is being set, some servers may not handle the entire length provided and may truncate the value.
- DB2 for OS/390 servers support up to a length of 32 characters.

- To ensure that the data is converted correctly when transmitted to a DRDA server, use only the characters A to Z, 0 to 9, and the underscore(_) or period (.).

Note: This is an IBM defined extension.

SQL_ATTR_INFO_USERID (DB2 CLI v6)

A null-terminated character string used to identify the client user ID sent to the host database server when using DB2 Connect. Please note:

- When the value is being set, some servers may not handle the entire length provided and may truncate the value.
- DB2 for OS/390 servers support up to a length of 16 characters.
- This user-id is not to be confused with the authentication user-id. This user-id is for identification purposes only and is not used for any authorization.
- To ensure that the data is converted correctly when transmitted to a DRDA server, use only the characters A to Z, 0 to 9, and the underscore(_) or period (.).

Note: This is an IBM defined extension.

SQL_ATTR_INFO_WRKSTNNAME (DB2 CLI v6)

A null-terminated character string used to identify the client workstation name sent to the host database server when using DB2 Connect. Please note:

- When the value is being set, some servers may not handle the entire length provided and may truncate the value.
- DB2 for OS/390 servers support up to a length of 18 characters.
- To ensure that the data is converted correctly when transmitted to a DRDA server, use only the characters A to Z, 0 to 9, and the underscore(_) or period (.).

Note: This is an IBM defined extension.

SQL_ATTR_LOGIN_TIMEOUT (DB2 CLI v2)

This connection attribute is defined by ODBC, but is not supported by DB2 CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

A 32-bit integer value corresponding to the number of seconds to wait for a login request to complete before returning to the application. The only permitted value for the *ValuePtr* argument is **0**, which means the connection attempt will wait until either a connection is established or the underlying communication layer times out.

SQL_ATTR_LONGDATA_COMPAT (DB2 CLI v2)

A 32-bit integer value indicating whether the character, double byte character and binary large object data types should be reported respectively as SQL_LONGVARCHAR, SQL_LONGVARGRAPHIC or SQL_LONGBINARY, enabling existing applications to access large object data types seamlessly. The option values are:

- **SQL_LD_COMPAT_NO**: The large object data types are reported as themselves (SQL_BLOB, SQL_CLOB, SQL_DBCLOB). This is the default.
- SQL_LD_COMPAT_YES: The large object data types (BLOB, CLOB and DBCLOB) are mapped to SQL_LONGVARBINARY, SQL_LONGVARCHAR and SQL_LONVARGRAPHIC; `SQLGetTypeInfo()` returns one entry each for SQL_LONGVARBINARY SQL_LONGVARCHAR.

Note: This is an IBM defined extension.

SQL_ATTR_MAXCONN (DB2 CLI v2)

A 32-bit integer value corresponding to the maximum concurrent connections that an application may desire to set up. The default value is **0**, which means no maximum - the application is allowed to set up as many connections as the system resources permit. The integer value must be 0 or a positive number.

This can be used as a governor for the maximum number of connections on a per application basis.

On OS/2, Windows 95, and Windows NT, if the NetBIOS protocol is in use, this value corresponds to the number of connections (NetBIOS sessions) that will be concurrently set up by the application. The range of values for OS/2 NetBIOS is 1 to 254. Specifying 0 (the default) will result in **5** *reserved* connections. *Reserved NetBIOS sessions* cannot be used by other applications. The number of connections specified by this parameter will be applied to any adaptor that the DB2 NetBIOS protocol uses to connect to the remote server (adapter number is specified in the node directory for a NetBIOS node).

The value that is in effect when the first connection is established is the value that will be used. Once the first connection has been established, attempts to change this value will be rejected. We recommended that the application set SQL_ATTR_MAXCONN at the environment level rather then on a connection basis. ODBC applications must set this attribute at the connection level since `SQLSetEnvAttr()` is not supported in ODBC.

The maximum concurrent connections can also be set using the MAXCONN DB2 CLI/ODBC configuration keyword. See “Configuring db2cli.ini” on page 159 for more information.

Note: This is an IBM defined extension.

SQL_ATTR_METADATA_ID (DB2 CLI v5)

This connection attribute is defined by ODBC, but is not supported by DB2 CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

An SQLUINTEGER value that determines how the string arguments of catalog functions are treated.

SQL_ATTR_ODBC_CURSORS (DB2 CLI v5)

This connection attribute is defined by ODBC, but is not supported by DB2 CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

A 32-bit option specifying how the Driver Manager uses the ODBC cursor library.

SQL_ATTR_PACKET_SIZE (DB2 CLI v5)

This connection attribute is defined by ODBC, but is not supported by DB2 CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

A 32-bit integer value specifying the network packet size in bytes.

SQL_ATTR_QUIET_MODE (DB2 CLI v2)

A 32-bit platform specific window handle.

If the application has never made a call to `SQLSetConnectOption()` with this option, then DB2 CLI would return a null parent window handle on `SQLGetConnectOption()` for this option and use a null parent window handle to display dialogue boxes. For example, if the end user has asked for (via an entry in the DB2 CLI initialization file) optimizer information to be displayed, DB2 CLI would display the dialogue box containing this information using a null window handle. (For some platforms, this means the dialogue box would be centered in the middle of the screen.)

If *ValuePtr* is a null pointer, then DB2 CLI does not display any dialogue boxes. In the above example where the end user has asked for the optimizer estimates to be displayed, DB2 CLI would not display these estimates because the application explicitly wants to suppress all such dialogue boxes.

If *ValuePtr* is not a null pointer, then it should be the parent window handle of the application. DB2 CLI uses this handle to display dialogue boxes. (For some platforms, this means the dialogue box would be centered with respect to the active window of the application.)

Note: This connection option cannot be used to suppress the `SQLDriverConnect()` dialogue box (which can be suppressed by setting the *fDriverCompletion* argument to SQL_DRIVER_NOPROMPT).

SQL_ATTR_SYNC_POINT (DB2 CLI v2)

A 32-bit integer value that allows the application to choose between one-phase coordinated transactions and two-phase coordinated transactions. The possible values are:

- **SQL_ONEPHASE**: One-phase commit is used to commit the work done by each database in a multiple database transaction. To ensure data integrity, each transaction must not have more than one database updated. The first database that has updates performed in a transaction becomes the only updater in that transaction, all other databases accessed are treated as read-only. Any update attempts to these read-only database within this transaction are rejected. This is the default.
- SQL_TWOPHASE: Two-phase commit is used to commit the work done by each database in a multiple database transaction. This requires the use of a Transaction Manager to coordinate two phase commits amongst the databases that support this protocol. Multiple readers and multiple updaters are allowed within a transaction.

Refer to the *SQL Reference* for more information on distributed unit of work (transactions).

All the connections within an application must have the same SQL_ATTR_CONNECTTYPE and SQL_SYNCPOINT values. The first connection determines the acceptable attributes for the subsequent connections. We recommend that the application set the SQL_ATTR_CONNECTTYPE attribute at the environment level rather than on a per connection basis. ODBC applications written to take advantage of coordinated DB2 transactions must set these attributes at the connection level as `SQLSetEnvAttr()` is not supported in ODBC.

The type of coordinated transaction can also be set using the SYNCPOINT DB2 CLI/ODBC configuration keyword. See "Configuring db2cli.ini" on page 159 for more information.

Note: This is an IBM extension. In embedded SQL, there is an additional sync point setting called SYNCPOINT NONE. This is more restrictive than the SQL_CONCURRENT_TRANS setting of the SQL_ATTR_CONNECTTYPE option because SYNCPOINT NONE does not allow for multiple connections to the same database. As a result, it is not necessary for DB2 CLI to support SYNCPOINT NONE.

SQL_ATTR_TRACE (DB2 CLI v5)

This connection attribute is defined by ODBC, but is not supported by DB2 CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

A 32-bit integer value telling DB2 CLI whether to perform tracing.

Instead of using the attribute, the DB2 CLI trace facility can be set using the TRACE DB2 CLI/ODBC configuration keyword. See "Configuring db2cli.ini" on page 159 for more information.

SQL_ATTR_TRACEFILE (DB2 CLI v5)

This connection attribute is defined by ODBC, but is not supported by DB2 CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

A null-terminated character string containing the name of the trace file.

Instead of using this attribute, the DB2 CLI trace file name is set using the TRACEFILENAME DB2 CLI/ODBC configuration keyword. See "Configuring db2cli.ini" on page 159 for more information.

SQL_ATTR_TRANSLATE_LIB (DB2 CLI v5)

This connection attribute is defined by ODBC, but is only supported by DB2 CLI on Windows 3.1. Any attempt to set or get this attribute on other platforms will result in an SQLSTATE of HYC00 (Driver not capable).

Indicate the directory where the DB2 Client Application Enabler for Windows or the Software Developers Kit for Windows has been installed. DB2TRANS.DLL is the DLL that contains codepage mapping tables.

This option is used on 16-bit versions of Windows when connecting to DB2 for OS/2 Version 1, or when using a version of DDCS for OS/2 prior to Version 2.3 in conjunction with the TRANSLATEOPTION, to provide proper mapping of NLS SBCS characters (such as the umlaut character in German) to the corresponding characters in the Windows codepage 1004.

Note: This option is useful when a Windows application connects to a downlevel server that does not support unequal codepage conversion (such as DB2 Version 1).

SQL_ATTR_TRANSLATE_OPTION (DB2 CLI v5)

This connection attribute is defined by ODBC, but is only supported by DB2 CLI on Windows 3.1. Any attempt to set or get this attribute on other platforms will result in an SQLSTATE of HYC00 (Driver not capable).

Defines the codepage number of the database in DB2 Version 1 (it can be obtained by querying the database configuration parameters). Specifying TRANSLATEDLL and TRANSLATEOPTION enables the translation of characters from codepage number *database codepage number* to the Windows 1004 codepage.

There are two supported values for *database codepage number*: 437 and 850. If you specify any other values, a warning is returned on the connect request indicating that translation is not possible.

Note: This option is useful when a Windows application connects to a downlevel server that does not support unequal codepage conversion (such as DB2 Version 1).

SQL_ATTR_TXN_ISOLATION (DB2 CLI v2)

A 32-bit bitmask that sets the transaction isolation level for the current connection referenced by *ConnectionHandle*. The valid values for *ValuePtr* can be determined at runtime by calling `SQLGetInfo()` with *fInfoType* set to SQL_TXN_ISOLATION_OPTIONS. The following values are accepted by DB2 CLI, but each server may only support a subset of these isolation levels:

- SQL_TXN_READ_UNCOMMITTED - Dirty reads, reads that cannot be repeated, and phantoms are possible.
- **SQL_TXN_READ_COMMITTED** - Dirty reads are not possible. Reads that cannot be repeated, and phantoms are possible.

  This is the default.
- SQL_TXN_REPEATABLE_READ - Dirty reads and reads that cannot be repeated are not possible. Phantoms are possible.
- SQL_TXN_SERIALIZABLE - Transactions can be serialized. Dirty reads, non-repeatable reads, and phantoms are not possible.
- SQL_TXN_NOCOMMIT - Any changes are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed. This is analogous to autocommit. This is not an SQL92 isolation level, but an IBM defined extension, supported only by DB2 Universal Database for AS/400.

In IBM terminology,

- SQL_TXN_READ_UNCOMMITTED is Uncommitted Read;
- SQL_TXN_READ_COMMITTED is Cursor Stability;
- SQL_TXN_REPEATABLE_READ is Read Stability;

- SQL_TXN_SERIALIZABLE is Repeatable Read.

For a detailed explanation of Isolation Levels, refer to the *SQL Reference*.

This option cannot be specified while there is an open cursor on any hstmt, or an outstanding transaction for this connection; otherwise, SQL_ERROR is returned on the function call (SQLSTATE **S1**011).

This attribute (or corresponding keyword) is only applicable if the default isolation level is used. If the application has specifically set the isolation level then this attribute will have no effect.

Note: There is an IBM extension that permits the setting of transaction isolation levels on a per statement handle basis. See the SQL_STMTTXN_ISOLATION option in the function description for `SQLSetStmtOption()`.

SQL_ATTR_WCHARTYPE (DB2 CLI v2)

A 32-bit integer that specifies, in a double-byte environment, which wchar_t (SQLDBCHAR) character format you want to use in your application. This option provides you the flexibility to choose between having your wchar_t data in multi-byte format or in wide-character format. There two possible values for this option:

- SQL_WCHARTYPE_CONVERT: character codes are converted between the graphic SQL data in the database and the application application variable. This allows your application to fully exploit the ANSI C mechanisms for dealing with wide character strings (L-literals, 'wc' string functions, etc.) without having to explicitly convert the data to multi-byte format before communicating with the database. The disadvantage is that the implicit conversions may have an impact on the runtime performance of your application, and may increase memory requirements. If you want WCHARTYPE CONVERT behavior 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.
- **SQL_WCHARTYPE_NOCONVERT**: no implicit character code conversion occurs between the application and the database. Data in the application variable is sent to and received from the database as unaltered DBCS characters. This allows the application to have improved performance, but the disadvantage is that the application must either refrain from using wide-character data in wchar_t (SQLDBCHAR) application variables, or it must explicitly call the

wcstombs() and mbstowcs() ANSI C functions to convert the data to and from multi-byte format when exchanging data with the database.

This is the default.

For additional information on the use of multi-byte application variables, refer to the *Application Development Guide*

Note: This is an IBM defined extension.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

DB2 CLI can return SQL_SUCCESS_WITH_INFO to provide information about the result of setting an option.

When *Attribute* is a statement attribute, `SQLSetConnectAttr()` can return any SQLSTATEs returned by `SQLSetStmtAttr()`.

Table 151. SQLSetConnectAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	General error.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed.	DB2 CLI did not support the value specified in **ValuePtr* and substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
08002	Connection in use.	The argument *Attribute* was SQL_ATTR_ODBC_CURSORS and DB2 CLI was already connected to the data source.
08003	Connection is closed.	An *Attribute* value was specified that required an open connection, but the *ConnectionHandle* was not in a connected state.
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was connected failed before the function completed processing.
24000	Invalid cursor state.	The argument *Attribute* was SQL_ATTR_CURRENT_QUALIFIER and a result set was pending.

Table 151. SQLSetConnectAttr SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec() in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY009	Invalid argument value.	A null pointer was passed for *ValuePtr* and the value in **ValuePtr* was a string value.
HY010	Function sequence error.	An asynchronously executing function was called for a *StatementHandle* associated with the *ConnectionHandle* and was still executing when SQLSetConnectAttr() was called. SQLExecute() or SQLExecDirect()was called for a *StatementHandle* associated with the *ConnectionHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. SQLBrowseConnect() was called for the *ConnectionHandle* and returned SQL_NEED_DATA. This function was called before SQLBrowseConnect() returned SQL_SUCCESS_WITH_INFO orSQL_SUCCESS.
HY011	Operation invalid at this time.	The argument *Attribute* was SQL_ATTR_TXN_ISOLATION and a transaction was open.
HY024	Invalid attribute value.	Given the specified *Attribute* value, an invalid value was specified in **ValuePtr*. (DB2 CLI returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE or SQL_ATTR_ASYNC_ENABLE. For all other connection and statement attributes, DB2 CLI must verify the value specified in *ValuePtr*.) The *Attribute* argument was SQL_ATTR_TRACEFILE or SQL_ATTR_TRANSLATE_LIB, and **ValuePtr* was an empty string.
HY090	Invalid string or buffer length.	The *StringLength* argument was less than 0, but was not SQL_NTS.
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid for this version of DB2 CLI.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid connection or statement attribute for the version of the DB2 CLI driver, but was not supported by the data source.

Restrictions

None.

Example

See `SQLConnect()`.

References

- "SQLGetConnectAttr - Get Current Attribute Setting" on page 439
- "SQLGetStmtAttr - Get Current Setting of a Statement Attribute" on page 545
- "SQLSetStmtAttr - Set Options Related to a Statement" on page 702
- "SQLAllocHandle - Allocate Handle" on page 220

SQLSetConnection - Set Connection Handle

Purpose

Specification:	DB2 CLI 2.1		

This function is needed if the application needs to deterministically switch to a particular connection before continuing execution. It should only be used when the application is mixing DB2 CLI function calls with embedded SQL function calls and multiple connections are involved.

Syntax

```
SQLRETURN   SQLSetConnection (SQLHDBC          ConnectionHandle); /* hdbc */
```

Function Arguments

Table 152. SQLSetConnection Arguments

Data Type	Argument	Use	Description
SQLHDBC	ConnectionHandle	input	The connection handle associated with the connection the application wishes to switch to.

Usage

In DB2 CLI version 1 it was possible to mix DB2 CLI calls with calls to routines containing embedded SQL as long as the connect request was issued via the DB2 CLI connect function. The embedded SQL routine would simply use the existing DB2 CLI connection.

Although this is still true, there is a potential complication: DB2 CLI allows multiple concurrent connections. This means that it is no longer clear which connection an embedded SQL routine would use upon being invoked. In practice, the embedded routine would use the connection associated with the most recent network activity. However, from the application's perspective, this is not always deterministic and it is difficult to keep track of this information. `SQLSetConnection()` is used to allow the application to *explicitly* specify which connection is active. The application can then call the embedded SQL routine.

`SQLSetConnection()` is not needed at all if the application makes purely DB2 CLI calls. This is because each statement handle is implicitly associated with a connection handle and there is never any confusion as to which connection a particular DB2 CLI function applies.

For more information on using embedded SQL within DB2 CLI applications refer to "Mixing Embedded SQL and DB2 CLI" on page 136.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 153. SQLSetConnection SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection is closed.	The connection handle provided is not currently associated with an open connection to a database server.
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no implementation defined SQLSTATE was defined. The error message returned by SQLError in the argument *szErrorMsg* describes the error and its cause.

Restrictions

None.

Example

Refer to “Mixed Embedded SQL and DB2 CLI Example” on page 137.

References

- “SQLConnect - Connect to a Data Source” on page 323
- “SQLDriverConnect - (Expanded) Connect to a Data Source” on page 350

SQLSetConnectOption - Set Connection Option

Deprecated

Note:

In ODBC version 3, SQLSetConnectOption() has been deprecated and replaced with SQLSetConnectAttr(); see "SQLSetConnectAttr - Set Connection Attributes" on page 618 for more information.

Although this version of DB2 CLI continues to support SQLSetConnectOption(), we recommend that you begin using SQLSetConnectAttr() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Note: This deprecated function cannot be used in a 64-bit environment. For more details see "Deprecated Functions Not Supported in a 64-bit Environment" on page 768.

Migrating to the New Function

The statement:

```
SQLSetConnectOption(
           *hdbc,
           SQL_AUTOCOMMIT,
           SQL_AUTOCOMMIT_OFF);
```

for example, would be rewritten using the new function as:

```
SQLSetConnectAttr(
           *hdbc,
           SQL_ATTR_AUTOCOMMIT,
           SQL_AUTOCOMMIT_OFF,
           0);
```

In versions of DB2 before DB2 Universal Database version 5, SQLSetConnectOption() could be used to set certain statement attributes as well as connection attributes. This behavior has since been removed; SQLSetConnectAttr() cannot be used to set statement attributes. See "Setting a Subset of Statement Attributes using SQLSetConnectAttr()" on page 770 for complete details on migrating a version 2 application that made use of this feature.

SQLSetCursorName - Set Cursor Name

Purpose

Specification:	DB2 CLI 1.1	ODBC 1.0	ISO CLI

SQLSetCursorName() associates a cursor name with the statement handle. This function is optional since DB2 CLI implicitly generates a cursor name when each statement handle is allocated.

Syntax

```
SQLRETURN   SQLSetCursorName (SQLHSTMT          StatementHandle,
                              SQLCHAR      FAR  *CursorName,
                              SQLSMALLINT       NameLength);
```

Function Arguments

Table 154. SQLSetCursorName Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLCHAR *	*CursorName*	input	Cursor name
SQLSMALLINT	*NameLength*	input	Length of contents of *CursorName* argument

Usage

DB2 CLI always generates and uses an internally generated cursor name when a query is prepared or executed directly. SQLSetCursorName() allows an application defined cursor name to be used in an SQL statement (a Positioned UPDATE or DELETE). DB2 CLI maps this name to the internal name. The name will remain associated with the statement handle, until the handle is dropped, or another SQLSetCursorName() is called on this statement handle.

Although SQLGetCursorName() will return the name set by the application (if one was set), error messages associated with positioned UPDATE and DELETE statements will refer to the internal name. For this reason, we recommend that you do not use SQLSetCursorName(), but instead use the internal name which can be obtained by calling SQLGetCursorName().

Cursor names must follow these rules:

- All cursor names within the connection must be unique.
- Each cursor name must be less than or equal to 18 bytes in length. Any attempt to set a cursor name longer than 18 bytes results in truncation of that cursor name to 18 bytes. (No warning is generated.)

- Since internally generated names begin with SQLCUR or SQL_CUR, the application must not input a cursor name starting with either SQLCUR or SQL_CUR in order to avoid conflicts with internal names.
- Since a cursor name is considered an identifier in SQL, it must begin with an English letter (a-z, A-Z) followed by any combination of digits (0-9), English letters or the underscore character (_).
- To permit cursor names containing characters other than those listed above (such as National Language Set or Double Bytes Character Set characters), the application must enclose the cursor name in double quotes (″).
- Unless the input cursor name is enclosed in double quotes, all leading and trailing blanks from the input cursor name string will be removed.

For efficient processing, applications should not include any leading or trailing spaces in the *CursorName* buffer. If the *CursorName* buffer contains a delimited identifier, applications should position the first double quote as the first character in the *CursorName* buffer.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 155. SQLSetCursorName SQLSTATEs

SQLSTATE	Description	Explanation
34000	Invalid cursor name.	The cursor name specified by the argument *CursorName* was invalid. The cursor name either begins with ″SQLCUR″ or ″SQL_CUR″ or violates the cursor naming rules (Must begin with a-z or A-Z followed by any combination of English letters, digits, or the '_' character. The cursor name specified by the argument *CursorName* already exists. The cursor name length is greater than the value returned by `SQLGetInfo()` with the SQL_MAX_CURSOR_NAME_LEN argument.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.

Table 155. SQLSetCursorName SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY009	Invalid argument value.	*CursorName* was a null pointer.
HY010	Function sequence error.	There is an open or positioned cursor on the statement handle. The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The argument *NameLength* was less than **0**, but not equal to SQL_NTS.

Authorization

None.

Example

```
/* From CLI sample setcurs.c */
/* ... */
    SQLCHAR * sqlstmt =
       "SELECT name, job FROM staff WHERE job = 'Clerk' FOR UPDATE OF job" ;
/* ... */
    /* allocate second statement handle for update statement */
    rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt2 ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

    /* Set Cursor for the SELECT statement's handle */
    rc = SQLSetCursorName(hstmt1, (SQLCHAR *)"JOBCURS", SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;

    rc = SQLExecDirect(hstmt1, sqlstmt, SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;

    /* bind name to first column in the result set */
    rc = SQLBindCol(hstmt1, 1, SQL_C_CHAR, (SQLPOINTER) name.s, 10,
                    &name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;

    /* bind job to second column in the result set */
    rc = SQLBindCol(hstmt1, 2, SQL_C_CHAR, (SQLPOINTER) job.s, 6,
                    &job.ind);
```

```
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;

printf("Job Change for all clerks\n");

while ((rc = SQLFetch(hstmt1)) == SQL_SUCCESS) {
    printf("Name: %-9.9s Job: %-5.5s \n", name.s, job.s);
    printf("Enter new job or return to continue\n");
    gets((char *)newjob);
    if (newjob[0] != '\0') {
        sprintf((char *)updstmt,
                "UPDATE staff set job = '%s' where current of JOBCURS",
                newjob);
        rc2 = SQLExecDirect(hstmt2, updstmt, SQL_NTS);
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt2, rc ) ;
    }
}
if (rc != SQL_NO_DATA_FOUND)
   CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc ) ;
```

References

- “SQLGetCursorName - Get Cursor Name” on page 444

SQLSetDescField - Set a Single Field of a Descriptor Record

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLSetDescField() sets the value of a single field of a descriptor record.

Syntax

```
SQLRETURN   SQLSetDescField  (SQLHDESC          DescriptorHandle,
                              SQLSMALLINT       RecNumber,
                              SQLSMALLINT       FieldIdentifier,
                              SQLPOINTER        ValuePtr,
                              SQLINTEGER        BufferLength);
```

Function Arguments

Table 156. SQLSetDescField Arguments

Data Type	Argument	Use	Description
SQLHDESC	*DescriptorHandle*	input	Descriptor handle.
SQLSMALLINT	*RecNumber*	input	Indicates the descriptor record containing the field that the application seeks to set. Descriptor records are numbered from 0, with record number 0 being the bookmark record. The *RecNumber* argument is ignored for header fields.
SQLSMALLINT	*FieldIdentifier*	input	Indicates the field of the descriptor whose value is to be set. For more information, see “FieldIdentifier Arguments” on page 659.
SQLPOINTER	ValuePtr	input	Pointer to a buffer containing the descriptor information, or a four-byte value. The data type depends on the value of *FieldIdentifier*. If *ValuePtr* is a four-byte value, either all four of the bytes are used, or just two of the four are used, depending on the value of the *FieldIdentifier* argument.

Table 156. SQLSetDescField Arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER	BufferLength	input	If *FieldIdentifier* is an ODBC-defined field and *ValuePtr* points to a character string or a binary buffer, this argument should be the length of **ValuePtr*. If *FieldIdentifier* is an ODBC-defined field and *ValuePtr* is an integer, *BufferLength* is ignored. If *FieldIdentifier* is a driver-defined field, the application indicates the nature of the field by setting the *BufferLength* argument. *BufferLength* can have the following values: • *ValuePtr* is a pointer to a character string, then *BufferLength* is the length of the string or SQL_NTS. • If *ValuePtr* is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in *BufferLength* • This places a negative value in *BufferLength*. • If *ValuePtr* is a pointer to a value other than a character string or a binary string, then *BufferLength* should have the value SQL_IS_POINTER. • If *ValuePtr* contains a fixed-length value, then *BufferLength* is either SQL_IS_INTEGER, SQL_IS_UINTEGER, SQL_IS_SMALLINT, or SQL_IS_USMALLINT, as appropriate.

Usage

An application can call `SQLSetDescField()` to set any descriptor field one at a time. One call to `SQLSetDescField()` sets a single field in a single descriptor. This function can be called to set any field in any descriptor type, provided the field can be set (see the table later in this section).

Note: If a call to `SQLSetDescField()` fails, the contents of the descriptor record identified by the *RecNumber* argument are undefined.

Other functions can be called to set multiple descriptor fields with a single call of the function. The `SQLSetDescRec()` function sets a variety of fields that affect the data type and buffer bound to a column or parameter (the TYPE, DATETIME_INTERVAL_CODE, OCTET_LENGTH, PRECISION, SCALE, DATA_PTR, OCTET_LENGTH_PTR, and INDICATOR_PTR fields) `SQLBindCol()` or `SQLBindParameter()` can be used to make a complete

specification for the binding of a column or parameter. These functions set a specific group of descriptor fields with one function call.

SQLSetDescField() can be called to change the binding buffersby adding an offset to the binding pointers (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, or SQL_DESC_OCTET_LENGTH_PTR). This changes the binding buffers without calling SQLBindCol()or SQLBindParameter(). This allows an application to change SQL_DESC_DATA_PTR without changing other fields, for instance SQL_DESC_DATA_TYPE.

Descriptor header fields are set by calling SQLSetDescField() with a *RecNumber* of 0, and the appropriate *FieldIdentifier*. Many header fields contain statement attributes, so may also be set by a call to SQLSetStmtAttr(). This allows applications to set a statement attribute without first obtaining a descriptor handle. A *RecNumber* of 0 is also used to set bookmark fields.

Note: The statement attribute SQL_ATTR_USE_BOOKMARKS should always be set before calling SQLSetDescField() to set bookmark fields. While this is not mandatory, it is strongly recommended.

Sequence of Setting Descriptor Fields

When setting descriptor fields by calling SQLSetDescField(), the application must follow a specific sequence:

1. The application must first set the SQL_DESC_TYPE, SQL_DESC_CONCISE_TYPE, or SQL_DESC_DATETIME_INTERVAL_CODE field.
2. After one of these fields has been set, the application can set an attribute of a data type, and the driver sets data type attribute fields to the appropriate default values for the data type. Automatic defaulting of type attribute fields ensures that the descriptor is always ready to use once the application has specified a data type. If the application explicitly sets a data type attribute, it is overriding the default attribute.
3. After one of the fields listed in Step 1 has been set, and data type attributes have been set, the application can set SQL_DESC_DATA_PTR. This prompts a consistency check of descriptor fields. If the application changes the data type or attributes after setting the SQL_DESC_DATA_PTR field, then the driver sets SQL_DESC_DATA_PTR to a null pointer, unbinding the record. This forces the application to complete the proper steps in sequence, before the descriptor record is usable.

Initialization of Descriptor Fields

When a descriptor is allocated, the fields in the descriptor can be initialized to a default value, be initialized without a default value, or be undefined for the type of descriptor. The following tables indicate the initialization of each field for each type of descriptor, with "D" indicating that the field is initialized with a default, and "ND" indicating that the field is initialized without a default. If a number is shown, the default value of the field is that number. The tables also indicate whether a field is read/write (R/W) or read-only (R).

The fields of an IRD have a default value only after the statement has been prepared or executed and the IRD has been populated, not when the statement handle or descriptor has been allocated. Until the IRD has been populated, any attempt to gain access to a field of an IRD will return an error.

Some descriptor fields are defined for one or more, but not all, of the descriptor types (ARDs and IRDs, and APDs and IPDs). When a field is undefined for a type of descriptor, it is not needed by any of the functions that use that descriptor. Because a descriptor is a logical view of data, rather than an actual data structure, these extra fields have no effect on the defined fields (for more information, see "Using Descriptors" on page 97).

The fields that can be accessed by `SQLGetDescField()` cannot necessarily be set by `SQLSetDescField()`. Fields that can be set by `SQLSetDescField()` are listed in the following tables.

The initialization of header fields is as follows:

Table 157. Initialization of Header Fields

SQL_DESC_ALLOC_TYPE (SQLSMALLINT)			
R/W:	ARD: R APD: R IRD: R IPD: R	Default:	ARD: SQL_DESC_ALLOC_AUTO for implicit or SQL_DESC_ALLOC_USER for explicit APD: SQL_DESC_ALLOC_AUTO for implicit or SQL_DESC_ALLOC_USER for explicit IRD: SQL_DESC_ALLOC_AUTO IPD: SQL_DESC_ALLOC_AUTO
SQL_DESC_ARRAY_SIZE (SQLUINTEGER)			

Table 157. Initialization of Header Fields (continued)

R/W:	ARD: R/W APD: R/W IRD: Unused IPD: Unused	Default:	ARD: [a] APD: [a] IRD: Unused IPD: Unused
SQL_DESC_ARRAY_STATUS_PTR (SQLUSMALLINT *)			
R/W:	ARD: R/W APD: R/W IRD: R/W IPD: R/W	Default:	ARD: Null ptr APD: Null ptr IRD: Null ptr IPD: Null ptr
SQL_DESC_BIND_OFFSET_PTR (SQLINTEGER *)			
R/W:	ARD: R/W APD: R/W IRD: Unused IPD: Unused	Default:	ARD: Null ptr APD: Null ptr IRD: Unused IPD: Unused
SQL_DESC_BIND_TYPE (SQLINTEGER)			
R/W:	ARD: R/W APD: R/W IRD: Unused IPD: Unused	Default:	ARD: SQL_BIND_BY_COLUMN APD: SQL_BIND_BY_COLUMN IRD: Unused IPD: Unused
SQL_DESC_COUNT (SQLSMALLINT)			
R/W:	ARD: R/W APD: R/W IRD: R IPD: R/W	Default:	ARD: 0 APD: 0 IRD: D IPD: 0
SQL_DESC_ROWS_PROCESSED_PTR (SQLUINTEGER *)			
R/W:	ARD: Unused APD: Unused IRD: R/W IPD: R/W	Default:	ARD: Unused APD: Unused IRD: Null Ptr IPD: Null Ptr

a These fields are defined only when the IPD is automatically populated

by DB2 CLI. If the fields are not automatically populated then they are undefined. If an application attempts to set these fields, SQLSTATE HY091 (Descriptor type out of range.) will be returned.

The initialization of record fields is as follows:

Table 158. Initialization of Record Fields

SQL_DESC_AUTO_UNIQUE_VALUE (SQLINTEGER)			
R/W:	ARD: Unused	Default:	ARD: Unused
	APD: Unused		APD: Unused
	IRD: R		IRD: D
	IPD: Unused		IPD: Unused
SQL_DESC_BASE_COLUMN_NAME (SQLCHAR *)			
R/W:	ARD: Unused	Default:	ARD: Unused
	APD: Unused		APD: Unused
	IRD: R		IRD: D
	IPD: Unused		IPD: Unused
SQL_DESC_BASE_TABLE_NAME (SQLCHAR *)			
R/W:	ARD: Unused	Default:	ARD: Unused
	APD: Unused		APD: Unused
	IRD: R		IRD: D
	IPD: Unused		IPD: Unused
SQL_DESC_CASE_SENSITIVE (SQLINTEGER)			
R/W:	ARD: Unused	Default:	ARD: Unused
	APD: Unused		APD: Unused
	IRD: R		IRD: D
	IPD: R		IPD: D [a]
SQL_DESC_CATALOG_NAME (SQLCHAR *)			
R/W:	ARD: Unused	Default:	ARD: Unused
	APD: Unused		APD: Unused
	IRD: R		IRD: D
	IPD: Unused		IPD: Unused
SQL_DESC_CONCISE_TYPE (SQLSMALLINT)			

Table 158. Initialization of Record Fields (continued)

R/W:	ARD: R/W	Default:	ARD: SQL_C_DEFAULT
	APD: R/W		APD: SQL_C_DEFAULT
	IRD: R		IRD: D
	IPD: R/W		IPD: ND

SQL_DESC_DATA_PTR (SQLPOINTER)

R/W:	ARD: R/W	Default:	ARD: Null ptr
	APD: R/W		APD: Null ptr
	IRD: Unused		IRD: Unused
	IPD: Unused		IPD: Unused [b]

SQL_DESC_DATETIME_INTERVAL_CODE (SQLSMALLINT)

R/W:	ARD: R/W	Default:	ARD: ND
	APD: R/W		APD: ND
	IRD: R		IRD: D
	IPD: R/W		IPD: ND

SQL_DESC_DATETIME_INTERVAL_PRECISION (SQLINTEGER)

R/W:	ARD: R/W	Default:	ARD: ND
	APD: R/W		APD: ND
	IRD: R		IRD: D
	IPD: R/W		IPD: ND

SQL_DESC_DISPLAY_SIZE (SQLINTEGER)

R/W:	ARD: Unused	Default:	ARD: Unused
	APD: Unused		APD: Unused
	IRD: R		IRD: D
	IPD: Unused		IPD: Unused

SQL_DESC_FIXED_PREC_SCALE (SQLSMALLINT)

R/W:	ARD: Unused	Default:	ARD: Unused
	APD: Unused		APD: Unused
	IRD: R		IRD: D
	IPD: R		IPD: D [a]

SQL_DESC_INDICATOR_PTR (SQLINTEGER *)

Table 158. Initialization of Record Fields (continued)

R/W:	ARD: R/W APD: R/W IRD: Unused IPD: Unused	Default:	ARD: Null ptr APD: Null ptr IRD: Unused IPD: Unused
SQL_DESC_LABEL (SQLCHAR *)			
R/W:	ARD: Unused APD: Unused IRD: R IPD: Unused	Default:	ARD: Unused APD: Unused IRD: D IPD: Unused
SQL_DESC_LENGTH (SQLUINTEGER)			
R/W:	ARD: R/W APD: R/W IRD: R IPD: R/W	Default:	ARD: ND APD: ND IRD: D IPD: ND
SQL_DESC_LITERAL_PREFIX (SQLCHAR *)			
R/W:	ARD: Unused APD: Unused IRD: R IPD: Unused	Default:	ARD: Unused APD: Unused IRD: D IPD: Unused
SQL_DESC_LITERAL_SUFFIX (SQLCHAR *)			
R/W:	ARD: Unused APD: Unused IRD: R IPD: Unused	Default:	ARD: Unused APD: Unused IRD: D IPD: Unused
SQL_DESC_LOCAL_TYPE_NAME (SQLCHAR *)			
R/W:	ARD: Unused APD: Unused IRD: R IPD: R	Default:	ARD: Unused APD: Unused IRD: D IPD: D [a]
SQL_DESC_NAME (SQLCHAR *)			

Table 158. Initialization of Record Fields (continued)

R/W:	ARD: Unused	Default:	ARD: ND
	APD: Unused		APD: ND
	IRD: R		IRD: D
	IPD: R/W		IPD: ND

SQL_DESC_NULLABLE (SQLSMALLINT)

R/W:	ARD: Unused	Default:	ARD: ND
	APD: Unused		APD: ND
	IRD: R		IRD: N
	IPD: R		IPD: ND

SQL_DESC_NUM_PREC_RADIX (SQLINTEGER)

R/W:	ARD: R/W	Default:	ARD: ND
	APD: R/W		APD: ND
	IRD: R		IRD: D
	IPD: R/W		IPD: ND

SQL_DESC_OCTET_LENGTH (SQLINTEGER)

R/W:	ARD: R/W	Default:	ARD: ND
	APD: R/W		APD: ND
	IRD: R		IRD: D
	IPD: R/W		IPD: ND

SQL_DESC_OCTET_LENGTH_PTR (SQLINTEGER *)

R/W:	ARD: R/W	Default:	ARD: Null ptr
	APD: R/W		APD: Null ptr
	IRD: Unused		IRD: Unused
	IPD: Unused		IPD: Unused

SQL_DESC_PARAMETER_TYPE (SQLSMALLINT)

R/W:	ARD: Unused	Default:	ARD: Unused
	APD: Unused		APD: Unused
	IPD: Unused		IPD: Unused
	IRD: R/W		IRD: D=SQL_PARAM_INPUT

SQL_DESC_PRECISION (SQLSMALLINT)

Table 158. Initialization of Record Fields (continued)

R/W:	ARD: R/W APD: R/W IRD: R IPD: R/W	Default:	ARD: ND APD: ND IRD: D IPD: ND

SQL_DESC_SCALE (SQLSMALLINT)

R/W:	ARD: R/W APD: R/W IRD: R IPD: R/W	Default:	ARD: ND APD: ND IRD: D IPD: ND

SQL_DESC_SCHEMA_NAME (SQLCHAR *)

R/W:	ARD: Unused APD: Unused IRD: R IPD: Unused	Default:	ARD: Unused APD: Unused IRD: D IPD: Unused

SQL_DESC_SEARCHABLE (SQLSMALLINT)

R/W:	ARD: Unused APD: Unused IRD: R IPD: Unused	Default:	ARD: Unused APD: Unused IRD: D IPD: Unused

SQL_DESC_TABLE_NAME (SQLCHAR *)

R/W:	ARD: Unused APD: Unused IRD: R IPD: Unused	Default:	ARD: Unused APD: Unused IRD: D IPD: Unused

SQL_DESC_TYPE (SQLSMALLINT)

R/W:	ARD: R/W APD: R/W IRD: R IPD: R/W	Default:	ARD: SQL_C_DEFAULT APD: SQL_C_DEFAULT IRD: D IPD: ND

SQL_DESC_TYPE_NAME (SQLCHAR *)

Table 158. Initialization of Record Fields (continued)

R/W:	ARD: Unused APD: Unused IRD: R IPD: R	Default:	ARD: Unused APD: Unused IRD: D IPD: D [a]
SQL_DESC_UNNAMED (SQLSMALLINT)			
R/W:	ARD: Unused APD: Unused IRD: R IPD: R/W	Default:	ARD: ND APD: ND IRD: D IPD: ND
SQL_DESC_UNSIGNED (SQLSMALLINT)			
R/W:	ARD: Unused APD: Unused IRD: R IPD: R	Default:	ARD: Unused APD: Unused IRD: D IPD: D [a]
SQL_DESC_UPDATABLE (SQLSMALLINT)			
R/W:	ARD: Unused APD: Unused IRD: R IPD: Unused	Default:	ARD: Unused APD: Unused IRD: D IPD: Unused

a These fields are defined only when the IPD is automatically populated by DB2 CLI. If the fields are not automatically populated then they are undefined. If an application attempts to set these fields, SQLSTATE HY091 (Descriptor type out of range.) will be returned.

b The SQL_DESC_DATA_PTR field in the IPD can be set to force a consistency check. In a subsequent call to `SQLGetDescField()` or `SQLGetDescRec()`, DB2 CLI is not required to return the value that SQL_DESC_DATA_PTR was set to.

FieldIdentifier Argument

The *FieldIdentifier* argument indicates the descriptor field to be set. A descriptor contains the descriptor header, consisting of the header fields described in the next section, and zero or more descriptor records, consisting of the record fields described in the following section.

Header Fields

Each descriptor has a header consisting of the following fields.

SQL_DESC_ALLOC_TYPE [All] This read-only SQLSMALLINT header field specifies whether the descriptor was allocated automatically by DB2 CLI or explicitly by the application. The application can obtain, but not modify, this field. The field is set to SQL_DESC_ALLOC_AUTO if the descriptor was automatically allocated. It is set to SQL_DESC_ALLOC_USER if the descriptor was explicitly allocated by the application.

SQL_DESC_ARRAY_SIZE [Application descriptors] In ARDs, this SQLUINTEGER header field specifies the number of rows in the rowset. This is the number of rows to be returned by a call to `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()`. The default value is 1. The field is also set through the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.

In APDs, this SQLUINTEGER header field specifies the number of values for each parameter.

The default value of this field is 1. If SQL_DESC_ARRAY_SIZE is greater than 1, SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR of the APD or ARD point to arrays. The cardinality of each array is equal to the value of this field.

This field in the ARD can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_ROWSET_SIZE attribute. This field in the APD can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_PARAMSET_SIZE attribute.

SQL_DESC_ARRAY_STATUS_PTR [All] For each descriptor type, this SQLUSMALLINT * header field points to an array of SQLUSMALLINT values. These arrays are named as follows:

- row status array (IRD)
- parameter status array (IPD)
- row operation array (ARD)
- parameter operation array (APD)

In the IRD, this header field points to a row status array containing status values after a call to `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()`. The array has as many elements as there are rows in the rowset. The application must allocate an array of SQLUSMALLINTs and set this field to point to the array. The field is set to a null pointer by default. DB2 CLI will populate the array, unless the SQL_DESC_ARRAY_STATUS_PTR field is set to a null pointer, in which case no status values are generated and the array is not populated.

Note: Behavior is undefined if the application sets the elements of the row status array pointed to by the SQL_DESC_ARRAY_STATUS_PTR field of the IRD. The array is initially populated by a call to SQLFetch(), SQLFetchScroll(), or SQLSetPos(). If the call did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the array pointed to by this field are undefined.

The elements in the array can contain the following values:

- SQL_ROW_SUCCESS: The row was successfully fetched and has not changed since it was last fetched.
- SQL_ROW_SUCCESS_WITH_INFO: The row was successfully fetched and has not changed since it was last fetched. However, a warning was returned about the row.
- SQL_ROW_ERROR: An error occurred while fetching the row.
- SQL_ROW_UPDATED: The row was successfully fetched and has been updated since it was last fetched. If the row is fetched again, its status is SQL_ROW_SUCCESS.
- SQL_ROW_DELETED: The row has been deleted since it was last fetched.
- SQL_ROW_ADDED: The row was inserted by SQLSetPos(). If the row is fetched again, its status is SQL_ROW_SUCCESS.
- SQL_ROW_NOROW: The rowset overlapped the end of the result set and no row was returned that corresponded to this element of the row status array.

This field in the ARD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_ROW_STATUS_PTR attribute.

In the IPD, this header field points to a parameter status array containing status information for each set of parameter values after a call to SQLExecute() or SQLExecDirect(). If the call to SQLExecute() or SQLExecDirect() did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the array pointed to by this field are undefined. The application must allocate an array of SQLUSMALLINTs and set this field to point to the array. The driver will populate the array, unless the SQL_DESC_ARRAY_STATUS_PTR field is set to a null pointer, in which case no status values are generated and the array is not populated.

The elements in the array can contain the following values:

- SQL_PARAM_SUCCESS: The SQL statement was successfully executed for this set of parameters.
- SQL_PARAM_SUCCESS_WITH_INFO: The SQL statement was successfully executed for this set of parameters; however, warning information is available in the diagnostics data structure.

- SQL_PARAM_ERROR: An error occurred in processing this set of parameters. Additional error information is available in the diagnostics data structure.
- SQL_PARAM_UNUSED: This parameter set was unused, possibly due to the fact that some previous parameter set caused an error that aborted further processing.
- SQL_PARAM_DIAG_UNAVAILABLE: Diagnostic information is not available. An example of this is when DB2 CLI treats arrays of parameters as a monolithic unit and so does not generate this level of error information.

This field in the APD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_PARAM_STATUS_PTR attribute.

In the ARD, this header field points to a row operation array of values that can be set by the application to indicate whether this row is to be ignored for SQLSetPos() operations.

The elements in the array can contain the following values:

- SQL_ROW_PROCEED: The row is included in the bulk operation using SQLSetPos(). (This setting does not guarantee that the operation will occur on the row. If the row has the status SQL_ROW_ERROR in the IRD row status array, DB2 CLI may not be able to perform the operation in the row.)
- SQL_ROW_IGNORE: The row is excluded from the bulk operation using SQLSetPos().

If no elements of the array are set, all rows are included in the bulk operation. If the value in the SQL_DESC_ARRAY_STATUS_PTR field of the ARD is a null pointer, all rows are included in the bulk operation; the interpretation is the same as if the pointer pointed to a valid array and all elements of the array were SQL_ROW_PROCEED. If an element in the array is set to SQL_ROW_IGNORE, the value in the row status array for the ignored row is not changed.

This field in the ARD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_ROW_OPERATION_PTR attribute.

In the APD, this header field points to a parameter operation array of values that can be set by the application to indicate whether this set of parameters is to be ignored when SQLExecute() or SQLExecDirect() is called. The elements in the array can contain the following values:

- SQL_PARAM_PROCEED: The set of parameters is included in the SQLExecute() or SQLExecDirect() call.

- SQL_PARAM_IGNORE: The set of parameters is excluded from the SQLExecute() or SQLExecDirect() call.

If no elements of the array are set, all sets of parameters in the array are used in the SQLExecute() or SQLExecDirect() calls. If the value in the SQL_DESC_ARRAY_STATUS_PTR field of the APD is a null pointer, all sets of parameters are used; the interpretation is the same as if the pointer pointed to a valid array and all elements of the array were SQL_PARAM_PROCEED.

This field in the APD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_PARAM_OPERATION_PTR attribute.

SQL_DESC_BIND_OFFSET_PTR [Application descriptors] This SQLINTEGER * header field points to the bind offset. It is set to a null pointer by default. If this field is not a null pointer, DB2 CLI dereferences the pointer and adds the dereferenced value to each of the deferred fields that has a non-null value in the descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR) at fetch time, and uses the new pointer values when binding.

The bind offset is always added directly to the values in the SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR fields. If the offset is changed to a different value, the new value is still added directly to the value in each descriptor field. The new offset is not added to the field value plus any earlier offset.

This field is a *deferred field*: it is not used at the time it is set, but is used at a later time by DB2 CLI to retrieve data.

This field in the ARD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_ROW_BIND_OFFSET_PTR attribute. This field in the ARD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_PARAM_BIND_OFFSET_PTR attribute.

See the description of row-wise binding in the “SQLFetchScroll” and “SQLBindParameter” sections.

SQL_DESC_BIND_TYPE [Application descriptors] This SQLINTEGER header field sets the binding orientation to be used for either binding columns or parameters.

In ARDs, this field specifies the binding orientation when SQLFetchScroll() is called on the associated statement handle.

To select column-wise binding for columns, this field is set to SQL_BIND_BY_COLUMN (the default).

This field in the ARD can also be set by calling SQLSetStmtAttr() with SQL_ATTR_ROW_BIND_TYPE Attribute.

In APDs, this field specifies the binding orientation to be used for dynamic parameters.

To select column-wise binding for parameters, this field is set to SQL_BIND_BY_COLUMN (the default).

This field in the APD can also be set by calling SQLSetStmtAttr() with SQL_ATTR_PARAM_BIND_TYPE Attribute.

SQL_DESC_COUNT [All] This SQLSMALLINT header field specifies the one-based index of the highest-numbered record that contains data. When DB2 CLI sets the data structure for the descriptor, it must also set the COUNT field to show how many records are significant. When an application allocates an instance of this data structure, it does not have to specify how many records to reserve room for. As the application specifies the contents of the records, DB2 CLI takes any required action to ensure that the descriptor handle refers to a data structure of the adequate size.

SQL_DESC_COUNT is not a count of all data columns that are bound (if the field is in an ARD), or all parameters that are bound (in an APD), but the number of the highest-numbered record. If a column or a parameter with a number that is less than the number of the highest-numbered column is unbound (by calling SQLBindCol() with the *Target ValuePtr* argument set to a null pointer, or SQLBindParameter() with the *Parameter ValuePtr* argument set to a null pointer), SQL_DESC_COUNT is not changed. If additional columns or parameters are bound with numbers greater than the highest-numbered record that contains data, DB2 CLI automatically increases the value in the SQL_DSEC_COUNT field. If all columns or parameters are unbound by calling SQLFreeStmt() with the SQL_UNBIND option, SQL_DESC_COUNT is set to 0.

The value in SQL_DESC_COUNT can be set explicitly by an application by calling SQLSetDescField(). If the value in SQL_DESC_COUNT is explicitly decreased, all records with numbers greater than the new value in SQL_DESC_COUNT are removed, unbinding the columns. If the value in SQL_DESC_COUNT is explicitly set to 0, and the field is in an APD, all parameter columns are unbound. If the value in SQL_DESC_COUNT is explicitly set to 0, and the field is in an ARD, all data buffers except a bound bookmark column are released.

The record count in this field of an ARD does not include a bound bookmark column.

SQL_DESC_ROWS_PROCESSED_PTR [Implementation descriptors] In an IRD, this SQLUINTEGER * header field points to a buffer containing the number of rows fetched after a call to SQLFetch() or SQLFetchScroll(), or the number of rows affected in a bulk operation performed by a call to SQLSetPos().

In an IPD, this SQLUINTEGER * header field points to a buffer containing the number of the row as each row of parameters is processed. No row number will be returned if this is a null pointer.

SQL_DESC_ROWS_PROCESSED_PTR is valid only after SQL_SUCCESS or SQL_SUCCESS_WITH_INFO has been returned after a call to SQLFetch() or SQLFetchScroll()(for an IRD field) or SQLExecute() or SQLExecDirect() (for an IPD field). If the return code is not one of the above, the location pointed to by SQL_DESC_ROWS_PROCESSED_PTR is undefined. If the call that fills in the buffer pointed to by this field did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined, unless it returns SQL_NO_DATA, in which case the value in the buffer is set to 0.

This field in the ARD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_ROWS_FETCHED_PTR attribute. This field in the ARD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_PARAMS_PROCESSED_PTR attribute.

The buffer pointed to by this field is allocated by the application. It is a deferred output buffer that is set by DB2 CLI. It is set to a null pointer by default.

Record Fields

Each descriptor contains one or more records consisting of fields that define either column data or dynamic parameters, depending on the type of descriptor. Each record is a complete definition of a single column or parameter.

SQL_DESC_AUTO_UNIQUE_VALUE [IRDs] This read-only SQLINTEGER record field contains SQL_TRUE if the column is an auto-incrementing column, or SQL_FALSE if the column is not an auto-incrementing column. This field is read-only, but the underlying auto-incrementing column is not necessarily read-only.

SQL_DESC_BASE_COLUMN_NAME [IRDs] This read-only SQLCHAR record field contains the base column name for the result set column. If a base

column name does not exist (as in the case of columns that are expressions), then this variable contains an empty string.

SQL_DESC_BASE_TABLE_NAME [IRDs] This read-only SQLCHAR record field contains the base table name for the result set column. If a base table name cannot be defined or is not applicable, then this variable contains an empty string.

SQL_DESC_CASE_SENSITIVE [Implementation descriptors] This read-only SQLINTEGER record field contains SQL_TRUE if the column or parameter is treated as case-sensitive for collations and comparisons, or SQL_FALSE if the column is not treated as case-sensitive for collations and comparisons, or if it is a non-character column.

SQL_DESC_CATALOG_NAME [IRDs] This read-only SQLCHAR record field contains the catalog or qualifier name for the base table that contains the column. The return value is driver-dependent if the column is an expression or if the column is part of a view. If the data source does not support catalogs (or qualifiers) or the catalog or qualifier name cannot be determined, this variable contains an empty string.

SQL_DESC_CONCISE_TYPE [All] This SQLSMALLINT header field specifies the concise data type for all data types, including the datetime and interval data types.

The values in the SQL_DESC_CONCISE_TYPE and SQL_DESC_TYPE fields are interdependent. Each time one of the fields is set, the other must also be set. SQL_DESC_CONCISE_TYPE can be set by a call to `SQLBindCol()` or `SQLBindParameter()`, or `SQLSetDescField()`. SQL_DESC_TYPE can be set by a call to `SQLSetDescField()` or `SQLSetDescRec()`.

If SQL_DESC_CONCISE_TYPE is set to a concise data type other than an interval or datetime data type, the SQL_DESC_TYPE field is set to the same value, and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to 0.

If SQL_DESC_CONCISE_TYPE is set to the concise datetime or interval data type, the SQL_DESC_TYPE field is set to the corresponding verbose type (SQL_DATETIME or SQL_INTERVAL), and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to the appropriate subcode.

SQL_DESC_DATA_PTR [Application descriptors and IPDs] This SQLPOINTER record field points to a variable that will contain the parameter value (for APDs) or the column value (for ARDs). The descriptor record (and either the column or parameter that it represents) is unbound if Target*ValuePtr* in a call to either `SQLBindCol()` or `SQLBindParameter()` is a null pointer, or the

SQL_DESC_DATA_PTR field in a call to SQLSetDescField() or SQLSetDescRec() is set to a null pointer. Other fields are not affected if the SQL_DESC_DATA_PTR field is set to a null pointer. If the call to SQLFetch() or SQLFetchScroll()that fills in the buffer pointed to by this field did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

This field is a deferred field: it is not used at the time it is set, but is used at a later time by DB2 CLI to retrieve data.

Whenever the SQL_DESC_DATA_PTR field is set, DB2 CLI checks that the value in the SQL_DESC_TYPE field contains a the valid DB2 CLI or ODBC data types, and that all other fields affecting the data types are consistent. See "Consistency Checks" on page 674.

SQL_DESC_DATETIME_INTERVAL_CODE [All] This SQLSMALLINT record field contains the subcode for the specific datetime data type when the SQL_DESC_TYPE field is SQL_DATETIME. This is true for both SQL and C data types.

This field can be set to the following for datetime data types:

Table 159. Datetime Subcodes

Datetime types	DATETIME_INTERVAL_CODE
SQL_TYPE_DATE / SQL_C_TYPE_DATE	SQL_CODE_DATE
SQL_TYPE_TIME / SQL_C_TYPE_TIME	SQL_CODE_TIME
SQL_TYPE_TIMESTAMP / SQL_C_TYPE_TIMESTAMP	SQL_CODE_TIMESTAMP

This field can also be set to other values (not listed here) for interval data types, which DB2 CLI does not support.

SQL_DESC_DATETIME_INTERVAL_PRECISION [All] This SQLINTEGER record field contains the interval leading precision if the TYPE field is SQL_INTERVAL (which DB2 CLI does not support).

SQL_DESC_DISPLAY_SIZE [IRDs] This read-only SQLINTEGER record field contains the maximum number of characters required to display the data from the column. The value in this field is not the same as the descriptor field LENGTH because the LENGTH field is undefined for all numeric types.

SQL_DESC_FIXED_PREC_SCALE [Implementation descriptors] This read-only SQLSMALLINT record field is set to SQL_TRUE if the column is an exact numeric column and has a fixed precision and non-zero scale (such as the MONEY data type), or SQL_FALSE if the column is not an exact numeric column with a fixed precision and scale.

SQL_DESC_INDICATOR_PTR [Application descriptors] In ARDs, this SQLINTEGER * record field points to the indicator variable. This variable contains SQL_NULL_DATA if the column value is a NULL. For APDs, the indicator variable is set to SQL_NULL_DATA to specify NULL dynamic arguments. Otherwise, the variable is zero (unless the values in SQL_DESC_INDICATOR_PTR and SQL_DESC_OCTET_LENGTH_PTR are the same pointer).

If the SQL_DESC_INDICATOR_PTR field in an ARD is a null pointer, DB2 CLI is prevented from returning information about whether the column is NULL or not. If the column is NULL and INDICATOR_PTR is a null pointer, SQLSTATE 22002, "Indicator variable required but not supplied," is returned when DB2 CLI attempts to populate the buffer after a call to `SQLFetch()` or `SQLFetchScroll()`. If the call to `SQLFetch()` or `SQLFetchScroll()` did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

The SQL_DESC_INDICATOR_PTR field determines whether the field pointed to by SQL_DESC_OCTET_LENGTH_PTR is set. If the data value for a column is NULL, DB2 CLI sets the indicator variable to SQL_NULL_DATA. The field pointed to by SQL_DESC_OCTET_LENGTH_PTR is then not set. If a NULL value is not encountered during the fetch, the buffer pointed to by SQL_DESC_INDICATOR_PTR is set to zero, and the buffer pointed to by SQL_DESC_OCTET_LENGTH_PTR is set to the length of the data.

If the INDICATOR_PTR field in an APD is a null pointer, the application cannot use this descriptor record to specify NULL arguments.

This field is a deferred field: it is not used at the time it is set, but is used at a later time by DB2 CLI to store data.

SQL_DESC_LABEL [IRDs] This read-only SQLCHAR record field contains the column label or title. If the column does not have a label, this variable contains the column name. If the column is unnamed and unlabeled, this variable contains an empty string.

SQL_DESC_LENGTH [All] This SQLUINTEGER record field is either the maximum or actual character length of a character string or a binary data type. It is the maximum character length for a fixed-length data type, or the actual character length for a variable-length data type. Its value always excludes the null termination character that ends the character string. Note that this field is a count of characters, not a count of bytes.

The value in this field may be different from the value for length defined in DB2 CLI Version 2.

SQL_DESC_LITERAL_PREFIX [IRDs] This read-only SQLCHAR record field contains the character or characters that DB2 CLI recognizes as a prefix for a literal of this data type. This variable contains an empty string for a data type for which a literal prefix is not applicable.

SQL_DESC_LITERAL_SUFFIX [IRDs] This read-only SQLCHAR record field contains the character or characters that DB2 CLI recognizes as a suffix for a literal of this data type. This variable contains an empty string for a data type for which a literal suffix is not applicable.

SQL_DESC_LOCAL_TYPE_NAME [Implementation descriptors] This read-only SQLCHAR record field contains any localized (native language) name for the data type that may be different from the regular name of the data type. If there is no localized name, then an empty string is returned. This field is for display purposes only.

SQL_DESC_NAME [Implementation descriptors] This SQLCHAR record field in a row descriptor contains the column alias, if it applies. If the column alias does not apply, the column name is returned. In either case, the UNNAMED field is set to SQL_NAMED. If there is no column name or a column alias, an empty string is returned in the NAME field and the UNNAMED field is set to SQL_UNNAMED.

An application can set the SQL_DESC_NAME field of an IPD to a parameter name or alias to specify stored procedure parameters by name. (The SQL_DESC_NAME field of an IRD is a read-only field; SQLSTATE HY091 (Invalid descriptor field identifier) will be returned if an application attempts to set it.

In IPDs, this field is undefined if dynamic parameters are not supported. If named parameters are supported and the version of DB2 CLI is capable of describing parameters, then the parameter name is returned in this field.

The column name value can be affected by the environment attribute SQL_ATTR_USE_LIGHT_OUTPUT_SQLDA. See “SQLSetEnvAttr - Set Environment Attribute” on page 682 for more information.

SQL_DESC_NULLABLE [Implementation descriptors] In IRDs, this read-only SQLSMALLINT record field is SQL_NULLABLE if the column can have NULL values; SQL_NO_NULLS if the column does not have NULL values; or SQL_NULLABLE_UNKNOWN if it is not known whether the column accepts NULL values. This field pertains to the result set column, not the base column.

In IPDs, this field is always set to SQL_NULLABLE, since dynamic parameters are always nullable, and cannot be set by an application.

SQL_DESC_NUM_PREC_RADIX [All] This SQLINTEGER field contains a value of 2 if the data type in the SQL_DESC_TYPE field is an approximate numeric data type, because the SQL_DESC_PRECISION field contains the number of bits. This field contains a value of 10 if the data type in the SQL_DESC_TYPE field is an exact numeric data type, because the SQL_DESC_PRECISION field contains the number of decimal digits. This field is set to 0 for all non-numeric data types.

SQL_DESC_OCTET_LENGTH [All] This SQLINTEGER record field contains the length, in bytes, of a character string or binary data type. For fixed-length character types, this is the actual length in bytes. For variable-length character or binary types, this is the maximum length in bytes. This value always excludes space for the null termination character for implementation descriptors and always includes space for the null termination character for application descriptors. For application data, this field contains the size of the buffer. For APDs, this field is defined only for output or input/output parameters.

SQL_DESC_OCTET_LENGTH_PTR [Application descriptors] This SQLINTEGER * record field points to a variable that will contain the total length in bytes of a dynamic argument (for parameter descriptors) or of a bound column value (for row descriptors).

For an APD, this value is ignored for all arguments except character string and binary; if this field points to SQL_NTS, the dynamic argument must be null-terminated. To indicate that a bound parameter will be a data-at-execute parameter, an application sets this field in the appropriate record of the APD to a variable that, at execute time, will contain the value SQL_DATA_AT_EXEC. If there is more than one such field, SQL_DESC_DATA_PTR can be set to a value uniquely identifying the parameter to help the application determine which parameter is being requested.

If the OCTET_LENGTH_PTR field of an ARD is a null pointer, DB2 CLI does not return length information for the column. If the SQL_DESC_OCTET_LENGTH_PTR field of an APD is a null pointer, DB2 CLI assumes that character strings and binary values are null terminated. (Binary values should not be null terminated, but should be given a length, in order to avoid truncation.)

If the call to `SQLFetch()` or `SQLFetchScroll()` that fills in the buffer pointed to by this field did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

This field is a deferred field: it is not used at the time it is set, but is used at a later time by DB2 CLI to buffer data.

By default this is a pointer to a 4-byte value. You can set the SQL_ATTR_USE_2BYTES_OCTET_LENGTH environment variable to change this. See “SQLSetEnvAttr - Set Environment Attribute” on page 682 for more details.

SQL_DESC_PARAMETER_TYPE [IPDs]This SQLSMALLINT record field is set to SQL_PARAM_INPUT for an input parameter, SQL_PARAM_INPUT_OUTPUT for an input/output parameter, or SQL_PARAM_OUTPUT for an output parameter. Set to SQL_PARAM_INPUT by default.

For an IPD, the field is set to SQL_PARAM_INPUT by default if the IPD is not automatically populated by DB2 CLI (the SQL_ATTR_ENABLE_AUTO_IPD statement attribute is SQL_FALSE). An application should set this field in the IPD for parameters that are not input parameters.

SQL_DESC_PRECISION [All] This SQLSMALLINT record field contains the number of digits for an exact numeric type, the number of bits in the mantissa (binary precision) for an approximate numeric type, or the numbers of digits in the fractional seconds component for the SQL_TYPE_TIME, SQL_TYPE_TIMESTAMP, or SQL_INTERVAL_SECOND data type. This field is undefined for all other data types.

The value in this field may be different from the value for precision defined in DB2 CLI Version 2.

SQL_DESC_SCALE [All] This SQLSMALLINT record field contains the defined scale for DECIMAL and NUMERIC data types. The field is undefined for all other data types.

The value in this field may be different from the value for scale defined in DB2 CLI Version 2. For more information, see Appendix D, “Data Types.”

SQL_DESC_SCHEMA_NAME [IRDs] This read-only SQLCHAR record field contains the schema name of the base table that contains the column. For many DBMS’s, this is the owner name. If the data source does not support schemas (or owners) or the schema name cannot be determined, this variable contains an empty string.

SQL_DESC_SEARCHABLE [IRDs] This read-only SQLSMALLINT record field is set to one of the following values:

- SQL_PRED_NONE if the column cannot be used in a WHERE clause. (This is the same as the SQL_UNSEARCEABLE value in DB2 CLI Version 2.)

- SQL_PRED_CHAR if the column can be used in a WHERE clause, but only with the LIKE predicate. (This is the same as the SQL_LIKE_ONLY value in DB2 Version 2.)
- SQL_PRED_BASIC if the column can be used in a WHERE clause with all the comparison operators except LIKE. (This is the same as the SQL_EXCEPT_LIKE value in DB2 CLI Version 2.)
- SQL_PRED_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.

SQL_DESC_TABLE_NAME [IRDs] This read-only SQLCHAR record field contains the name of the base table that contains this column.

SQL_DESC_TYPE [All] This SQLSMALLINT record field specifies the concise SQL or C data type for all data types except datetime and interval data types. For the datetime and interval data types, this field specifies the verbose data type, i.e., SQL_DATETIME or SQL_INTERVAL.

Whenever this field contains SQL_DATETIME or SQL_INTERVAL, the SQL_DESC_DATETIME_INTERVAL_CODE field must contain the appropriate subcode for the concise type. For datetime data types, SQL_DESC_TYPE contains SQL_DATETIME, and the SQL_DESC_DATETIME_INTERVAL_CODE field contains a subcode for the specific datetime data type. For interval data types, SQL_DESC_TYPE contains SQL_INTERVAL, and the SQL_DESC_DATETIME_INTERVAL_CODE field contains a subcode for the specific interval data type.

The values in the SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE fields are interdependent. Each time one of the fields is set, the other must also be set. SQL_DESC_TYPE can be set by a call to `SQLSetDescField()` or `SQLSetDescRec()`. SQL_DESC_CONCISE_TYPE can be set by a call to `SQLBindCol()` or `SQLBindParameter()`, or `SQLSetDescField()`.

If SQL_DESC_TYPE is set to a concise data type other than an interval or datetime data type, the SQL_DESC_CONCISE_TYPE field is set to the same value, and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to 0.

If SQL_DESC_TYPE is set to the verbose datetime or interval data type (SQL_DATETIME or SQL_INTERVAL), and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to the appropriate subcode, the SQL_DESC_CONCISE TYPE field is set to the corresponding concise type. Trying to set SQL_DESC_TYPE to one of the concise datetime or interval types will return SQLSTATE HY021 (Inconsistent descriptor information).

When the SQL_DESC_TYPE field is set by a call to SQLSetDescField(), the following fields are set to the following default values. The values of the remaining fields of the same record are undefined:

Table 160. Default Values

SQL_DESC_TYPE	Other fields Implicitly Set
SQL_CHAR, SQL_VARCHAR	SQL_DESC_LENGTH is set to 1. SQL_DESC_PRECISION is set to 0.
SQL_DECIMAL, SQL_NUMERIC	SQL_DESC_SCALE is set to 0. SQL_DESC_PRECISION is set to the precision for the respective data type.
SQL_FLOAT	SQL_DESC_PRECISION is set to the default precision for SQL_FLOAT.
SQL_DATETIME	This datatype is not supported by DB2 CLI.
SQL_INTERVAL	This datatype is not supported by DB2 CLI.

When an application calls SQLSetDescField() to set fields of a descriptor, rather than calling SQLSetDescRec(), the application must first declare the data type. If the values implicitly set are unacceptable, the application can then call SQLSetDescField() to set the unacceptable value explicitly.

SQL_DESC_TYPE_NAME [Implementation descriptors] This read-only SQLCHAR record field contains the data-source-dependent type name (for example, “CHAR”, “VARCHAR”, and so on). If the data type name is unknown, this variable contains an empty string.

SQL_DESC_UNNAMED [Implementation descriptors] This SQLSMALLINT record field in a row descriptor is set to either SQL_NAMED or SQL_UNNAMED. If the NAME field contains a column alias, or if the column alias does not apply, the UNNAMED field is set to SQL_NAMED. If there is no column name or a column alias, the UNNAMED field is set to SQL_UNNAMED.

An application can set the SQL_DESC_UNNAMED field of an IPD to SQL_UNNAMED. SQLSTATE HY091 (Invalid descriptor field identifier) is returned if an application attempts to set the SQL_DESC_UNNAMED field of an IPD to SQL_NAMED. The SQL_DESC_UNNAMED field of an IRD is read-only; SQLSTATE HY091 (Invalid descriptor field identifier) will be returned if an application attempts to set it.

SQL_DESC_UNSIGNED [Implementation descriptors] This read-only SQLSMALLINT record field is set to SQL_TRUE if the column type is unsigned or non-numeric, or SQL_FALSE if the column type is signed.

SQL_DESC_UPDATABLE [IRDs] This read-only SQLSMALLINT record field is set to one of the following values:

- SQL_ATTR_READ_ONLY if the result set column is read-only.

- SQL_ATTR_WRITE if the result set column is read-write.
- SQL_ATTR_READWRITE_UNKNOWN if it is not known whether the result set column is updatable or not.

SQL_DESC_UPDATABLE describes the updatability of the column in the result set, not the column in the base table. The updatability of the column in the base table on which this result set column is based may be different than the value in this field. Whether a column is updatable can be based on the data type, user privileges, and the definition of the result set itself. If it is unclear whether a column is updatable, SQL_UPDT_READWRITE_UNKNOWN should be returned.

Consistency Checks

A consistency check is performed by DB2 CLI automatically whenever an application passes in a value for the SQL_DESC_DATA_PTR field of the ARD, APD, or IPD. If any of the fields is inconsistent with other fields, `SQLSetDescField()` will return SQLSTATE HY021, "Inconsistent descriptor information." For more information see `SQLSetDescRec()`, "Consistency Checks" on page 679.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 161. SQLSetDescField SQLSTATEs

SQLSTATE	Description	Explanation
01000	General warning	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed.	DB2 CLI did not support the value specified in **ValuePtr* (if *ValuePtr* was a pointer) or the value in *ValuePtr* (if *ValuePtr* was a four-byte value), or **ValuePtr* was invalid because of SQL constraints or requirements, so DB2 CLI substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)

Table 161. SQLSetDescField SQLSTATEs (continued)

SQLSTATE	Description	Explanation
07009	Invalid descriptor index.	The *FieldIdentifier* argument was a header field, and the *RecNumber* argument was not 0. The *RecNumber* argument was 0 and the *DescriptorHandle* was an IPD. The *RecNumber* argument was less than 0.
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was connected failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The *DescriptorHandle* was associated with a StatementHandle for which an asynchronously executing function (not this one) was called and was still executing when this function was called. `SQLExecute()` or `SQLExecDirect()` was called for the *StatementHandle* with which the *DescriptorHandle* was associated and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY016	Cannot modify an implementation row descriptor.	The *DescriptorHandle* argument was associated with an IRD, and the *FieldIdentifier* argument was not SQL_DESC_ARRAY_STATUS_PTR.
HY021	Inconsistent descriptor information.	The TYPE field, or any other field associated with the TYPE field in the descriptor, was not valid or consistent. The TYPE field was not a valid DB2 CLI C type. Descriptor information checked during a consistency check was not consistent. (see “Consistency Checks” on page 674)
HY091	Descriptor type out of range.	The value specified for the *FieldIdentifier* argument was not a DB2 CLI defined field and was not a defined value. The value specified for the *RecNumber* argument was greater than the value in the SQL_DESC_COUNT field. The *FieldIdentifier* argument was SQL_DESC_ALLOC_TYPE.

Table 161. SQLSetDescField SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY092	Option type out of range.	The value specified for the *Attribute* argument was not valid.
HY105	Invalid parameter type.	The value specified for the SQL_DESC_PARAMETER_TYPE field was invalid. (For more information, see the *InputOutputType* Argument section in `SQLBindParameter()`.)

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLSetDescRec - Set Multiple Descriptor Fields for a Column or Parameter Data" on page 677
- "SQLGetDescField - Get Single Field Settings of Descriptor Record" on page 458
- "SQLGetDescRec - Get Multiple Field Settings of Descriptor Record" on page 463
- "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227
- "SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator" on page 247

SQLSetDescRec - Set Multiple Descriptor Fields for a Column or Parameter Data

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

The SQLSetDescRec() function sets multiple descriptor fields that affect the data type and buffer bound to a column or parameter data.

Syntax

```
SQLRETURN   SQLSetDescRec    (SQLHDESC          DescriptorHandle,
                              SQLSMALLINT       RecNumber,
                              SQLSMALLINT       Type,
                              SQLSMALLINT       SubType,
                              SQLINTEGER        Length,
                              SQLSMALLINT       Precision,
                              SQLSMALLINT       Scale,
                              SQLPOINTER        DataPtr,
                              SQLINTEGER        *StringLengthPtr,
                              SQLINTEGER        *IndicatorPtr);
```

Function Arguments

Table 162. SQLSetDescRec Arguments

Data Type	Argument	Use	Description
SQLHDESC	*DescriptorHandle*	input	Descriptor handle. This must not be an IRD handle.
SQLSMALLINT	*RecNumber*	input	Indicates the descriptor record that contains the fields to be set. Descriptor records are numbered from 0, with record number 0 being the bookmark record. This argument must be equal to or greater than 0. If *RecNumber* is greater than the value of SQL_DESC_COUNT, *RecNumber* is changed to the value of SQL_DESC_COUNT.
SQLSMALLINT	*Type*	input	The value to which to set the SQL_DESC_TYPE field for the descriptor record.
SQLSMALLINT	*SubType*	input	For records whose type is SQL_DATETIME or SQL_INTERVAL, this is the value to which to set the SQL_DESC_DATETIME_INTERVAL_CODE field.
SQLINTEGER	*Length*	input	The value to which to set the SQL_DESC_OCTET_LENGTH field for the descriptor record.

Table 162. SQLSetDescRec Arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	*Precision*	input	The value to which to set the PRECISION field for the descriptor record.
SQLSMALLINT	*Scale*	input	The value to which to set the SCALE field for the descriptor record.
SQLPOINTER	*DataPtr*	Deferred Input or Output	The value to which to set the SQL_DESC_DATA_PTR field for the descriptor record. *DataPtr* can be set to a null pointer to set the SQL_DESC_DATA_PTR field to a null pointer.
SQLINTEGER	*StringLengthPtr*	Deferred Input or Output	The value to which to set the SQL_DESC_OCTET_LENGTH_PTR field for the descriptor record. *StringLengthPtr* can be set to a null pointer to set the SQL_DESC_OCTET_LENGTH_PTR field to a null pointer.
SQLINTEGER	*IndicatorPtr*	Deferred Input or Output	The value to which to set the SQL_DESC_INDICATOR_PTR field for the descriptor record. *IndicatorPtr* can be set to a null pointer to set the SQL_DESC_INDICATOR_PTR field to a null pointer.

Usage

An application can call `SQLSetDescRec()` to set the following fields for a single column or parameter:

- SQL_DESC_TYPE
- SQL_DESC_OCTET_LENGTH
- SQL_DESC_PRECISION
- SQL_DESC_SCALE
- SQL_DESC_DATA_PTR
- SQL_DESC_OCTET_LENGTH_PTR
- SQL_DESC_INDICATOR_PTR

(SQL_DESC_DATETIME_INTERVAL_CODE is also defined by ODBC but is not supported by DB2 CLI.)

Note: If a call to `SQLSetDescRec()` fails, the contents of the descriptor record identified by the *RecNumber* argument are undefined.

When binding a column or parameter, `SQLSetDescRec()` allows you to change multiple fields affecting the binding without calling `SQLBindCol()` or

SQLBindParameter(), or making multiple calls to SQLSetDescField(). SQLSetDescRec() can set fields on a descriptor not currently associated with a statement. Note that SQLBindParameter() sets more fields than SQLSetDescRec(), can set fields on both an APD and an IPD in one call, and does not require a descriptor handle.

The statement attribute SQL_ATTR_USE_BOOKMARKS should always be set before calling SQLSetDescRec() with a *RecNumber* argument of 0 to set bookmark fields. While this is not mandatory, it is strongly recommended.

Consistency Checks

A consistency check is performed by DB2 CLI automatically whenever an application sets the SQL_DESC_DATA_PTR field of the APD, ARD, or IPD. Calling SQLSetDescRec() always prompts a consistency check. If any of the fields is inconsistent with other fields, SQLSetDescRec() will return SQLSTATE HY021, "Inconsistent descriptor information."

Application Descriptors

Whenever an application sets the SQL_DESC_DATA_PTR field of an APD, ARD, or IPD, DB2 CLI checks that the value of the SQL_DESC_TYPE field and the values applicable to that SQL_DESC_TYPE field are valid and consistent. This check is always perform when SQLBindParameter() or SQLBindCol() is called, or when SQLSetDescRec() is called for an APD, ARD, or IPD. This consistency check includes the following checks on application descriptor fields:

- The SQL_DESC_TYPE field must be one of the valid C or SQL types. The SQL_DESC_CONCISE_TYPE field must be one of the valid C or SQL types.
- If the SQL_DESC_TYPE field indicates a numeric type, the SQL_DESC_PRECISION and SQL_DESC_SCALE fields are verified to be valid.
- If the SQL_DESC_CONCISE_TYPE field is a time data type the SQL_DESC_PRECISION field is verified to be a valid seconds precision.

The SQL_DESC_DATA_PTR field of an IPD is not normally set; however, an application can do so to force a consistency check of IPD fields. A consistency check cannot be performed on an IRD. The value that the SQL_DESC_DATA_PTR field of the IPD is set to is not actually stored, and cannot be retrieved by a call to SQLGetDescField() or SQLGetDescRec(); the setting is made only to force the consistency check.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 163. SQLSetDescRec SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
07009	Invalid descriptor index.	The *RecNumber* argument was set to 0, and the *DescriptorHandle* was an IPD handle. The *RecNumber* argument was less than 0. The *RecNumber* argument was greater than the maximum number of columns or parameters that the data source can support, and the *DescriptorHandle* argument was an APD, IPD, or ARD. The *RecNumber* argument was equal to 0, and the *DescriptorHandle* argument referred to an implicitly allocated APD. (This error does not occur with an explicitly allocated application descriptor, because it is not known whether an explicitly allocated application descriptor is an APD or ARD until execute time.)
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was connected failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The *DescriptorHandle* was associated with a *StatementHandle* for which an asynchronously executing function (not this one) was called and was still executing when this function was called. `SQLExecute()` or `SQLExecDirect()` was called for the *StatementHandle* with which the *DescriptorHandle* was associated and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Table 163. SQLSetDescRec SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY016	Cannot modify an implementation row descriptor.	The *DescriptorHandle* argument was associated with an IRD.
HY021	Inconsistent descriptor information.	The *Type* field, or any other field associated with the TYPE field in the descriptor, was not valid or consistent. Descriptor information checked during a consistency check was not consistent. (See "Consistency Checks" on page 674 in `SQLSetDescField()`.)

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLSetDescField - Set a Single Field of a Descriptor Record" on page 649
- "SQLGetDescField - Get Single Field Settings of Descriptor Record" on page 458
- "SQLGetDescRec - Get Multiple Field Settings of Descriptor Record" on page 463
- "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227
- "SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator" on page 247

SQLSetEnvAttr - Set Environment Attribute

Purpose

Specification:	DB2 CLI 2.1		ISO CLI

SQLSetEnvAttr() sets an environment attribute for the current environment.

Syntax

```
SQLRETURN   SQLSetEnvAttr    (SQLHENV          EnvironmentHandle, /* henv */
                              SQLINTEGER       Attribute,
                              SQLPOINTER       ValuePtr,          /* Value */
                              SQLINTEGER       StringLength);
```

Function Arguments

Table 164. SQLSetEnvAttr Arguments

Data Type	Argument	Use	Description
SQLHENV	EnviornmentHandle	Input	Environment handle.
SQLINTEGER	Attribute	Input	Environment attribute to set, refer to "Environment Attributes" for the list of attributes and their descriptions.
SQLPOINTER	ValuePtr	Input	The desired value for *Attribute*.
SQLINTEGER	StringLength	Input	Length of *ValuePtr* in bytes if the attribute value is a character string; if *Attribute* does not denote a string, then DB2 CLI ignores *StringLength*.

Usage

Once set, the attribute's value affects all connections under this environment.

The application can obtain the current attribute value by calling SQLGetEnvAttr().

Environment Attributes

SQL_ATTR_CONNECTION_POOLING

32-bit integer value that enables or disables connection pooling at the environment level. The following values are used:

- **SQL_CP_OFF** = Connection pooling is turned off. This is the default.
- SQL_CP_ONE_PER_DRIVER = A single, global connection pool is supported for each DB2 CLI application. Every connection in a pool is associated with the application.

- SQL_CP_ONE_PER_HENV = A single connection pool is supported for each environment. Every connection in a pool is associated with one environment.

Connection pooling is enabled by calling SQLSetEnvAttr() to set the SQL_ATTR_CONNECTION_POOLING attribute to SQL_CP_ONE_PER_DRIVER or SQL_CP_ONE_PER_HENV. This call must be made before the application allocates the shared environment for which connection pooling is to be enabled. The environment handle in the call to SQLSetEnvAttr() is set to null, which makes SQL_ATTR_CONNECTION_POOLING a process-level attribute. After connection pooling is enabled, the application then allocates an implicit shared environment by calling SQLAllocHandle() with the *InputHandle* argument set to SQL_HANDLE_ENV.

After connection pooling has been enabled and a shared environment has been selected for an application, SQL_ATTR_CONNECTION_POOLING cannot be reset for that environment, since SQLSetEnvAttr() is called with a null environment handle when setting this attribute. If this attribute is set while connection pooling is already enabled on a shared environment, the attribute only affects shared environments that are allocated subsequently.

SQL_ATTR_CONNECTTYPE

A 32-bit integer value that specifies whether this application is to operate in a coordinated or uncoordinated distributed environment. If the processing needs to be coordinated, then this option must be considered in conjunction with the SQL_ATTR_SYNC_POINT connection option. The possible values are:

- **SQL_CONCURRENT_TRANS**: The application can have concurrent multiple connections to any one database or to multiple databases. Each connection has its own commit scope. No effort is made to enforce coordination of transaction. If an application issues a commit using the environment handle on SQLTransact() and not all of the connections commit successfully, the application is responsible for recovery.

  The current setting of the SQL_ATTR_SYNC_POINT attribute is ignored.

  This is the default.
- SQL_COORDINATED_TRANS: The application wishes to have commit and rollbacks coordinated among multiple database connections. This option setting corresponds to the specification of the Type 2 CONNECT in embedded SQL and must be considered in conjunction with the SQL_ATTR_SYNC_POINT connection

option. In contrast to the SQL_CONCURRENT_TRANS setting described above, the application is permitted only one open connection per database.

This attribute must be set before allocating any connection handles, otherwise, the SQLSetEnvAttr() call will be rejected.

All the connections within an application must have the same SQL_ATTR_CONNECTTYPE and SQL_ATTR_SYNC_POINT values. This attribute can also be set using the SQLSetConnectAttr function. We recommend that the application set the SQL_ATTR_CONNECTTYPE attribute at the environment level rather than on a per connection basis. ODBC applications written to take advantage of coordinated DB2 transactions must set these attributes at the connection level for each connection using SQLSetConnectAttr() as SQLSetEnvAttr() is not supported in ODBC.

Note: This is an IBM defined extension.

SQL_ATTR_CP_MATCH

A 32-bit value that determines how a connection is chosen from a connection pool. When SQLConnect() or SQLDriverConnect() is called, DB2 CLI (or the Driver Manager when used) determines which connection is reused from the pool. DB2 CLI (or the Driver Manager) attempts to match the connection options in the call and the connection attributes set by the application to the keywords and connection attributes of the connections in the pool. The value of this attribute determines the level of precision of the matching criteria.

The following values are used to set the value of this attribute:

- **SQL_CP_STRICT_MATCH** = Only connections that exactly match the connection options in the call and the connection attributes set by the application are reused. This is the default.
- SQL_CP_RELAXED_MATCH = Connections with matching connection string keywords can be used. Keywords must match, but not all connection attributes must match.

SQL_ATTR_MAXCONN

A 32-bit integer value corresponding to the number that maximum concurrent connections that an application may desire to set up. The default value is **0**, which means no maximum - the application is allowed to set up as many connections as the system resources permit. The integer value must be 0 or a positive number.

This can be used as a governor for the maximum number of connections on a per application basis.

On OS/2, if the NetBIOS protocol is in use, this value corresponds to the number of connections (NetBIOS sessions) that will be concurrently set up by the application. The range of values for OS/2 NetBIOS is 1 to 254. Specifying 0 (the default) will result in **5** *reserved* connections. *Reserved NetBIOS sessions* cannot be used by other applications. The number of connections specified by this parameter will be applied to any adaptor that the DB2 NetBIOS protocol uses to connect to the remote server (adapter number is specified in the node directory for a NetBIOS node).

The value that is in effect when the first connection is established is the value that will be used. Once the first connection has been established, attempts to change this value will be rejected. We recommended that the application set SQL_ATTR_MAXCONN at the environment level rather then on a connection basis. ODBC applications must set this attribute at the connection level since `SQLSetEnvAttr()` is not supported in ODBC.

Note: This is an IBM defined extension.

SQL_ATTR_ODBC_VERSION

A 32-bit integer that determines whether certain functionality exhibits ODBC 2.x (DB2 CLI v2) behavior or ODBC 3.0 (DB2 CLI v5) behavior.

It is recommended that all DB2 CLI applications set this environment attribute. ODBC applications must set this environment attribute before calling any function that has an SQLHENV argument, or the call will return SQLSTATE HY010 (Function sequence error.).

The following values are used to set the value of this attribute:

- SQL_OV_ODBC3: Causes the following ODBC 3.0 (DB2 CLI v5) behavior:
  - DB2 CLI returns and expects ODBC 3.0 (DB2 CLI v5) codes for date, time, and timestamp.
  - DB2 CLI returns ODBC 3.0(DB2 CLI v5) SQLSTATE codes when `SQLError()`, `SQLGetDiagField()`, or `SQLGetDiagRec()` are called.
  - The *CatalogName* argument in a call to `SQLTables()` accepts a search pattern.
- SQL_OV_ODBC2 Causes the following ODBC 2.x (DB2 CLI v2) behavior:
  - DB2 CLI returns and expects ODBC 2.x (DB2 CLI v2) codes for date, time, and timestamp.
  - DB2 CLI returns ODBC 2.0 DB2 CLI v2) SQLSTATE codes when `SQLError()`, `SQLGetDiagField()`, or `SQLGetDiagRec()` are called.
  - The *CatalogName* argument in a call to `SQLTables()` does not accept a search pattern.

SQL_ATTR_OUTPUT_NTS

A 32-bit integer value which controls the use of null-termination in output arguments. The possible values are:

- **SQL_TRUE**: DB2 CLI uses null termination to indicate the length of output character strings (default).

  This is the default.
- SQL_FALSE: DB2 CLI does not use null termination in output character strings.

The CLI functions affected by this attribute are all functions called for the environment (and for any connections and statements allocated under the environment) that have character string parameters.

This attribute can only be set when there are no connection handles allocated under this environment.

SQL_ATTR_PROCESSCTRL

A 32-bit mask that sets process level attributes which affect all environments and connections for the process. This attribute must be set before the environment handle is allocated.

The call to `SQLSetEntAttr()` must have the *EnvironmentHandle* argument set to SQL_NULL_HANDLE. The settings remain in effect for the life of the process. Generally this attribute is only used for performance sensitive applications, where large numbers of CLI function calls are being made. Before setting any of these bits, ensure that the application, and any other libraries that the application calls, comply with the restrictions listed.

The following values may be combined to form a bitmask:

- SQL_PROCESSCTL_NOTHREAD - This bit indicates that the application does not use multiple threads, or if it does use multiple threads, guarantees that all DB2 calls will be serialized by the application. If set, DB2 CLI does not make any system calls to serialize calls to CLI, and sets the DB2 context type to SQL_CTX_ORIGINAL.
- SQL_PROCESSCTL_NOFORK - This bit indicates that the application will never fork a child process. If set, DB2 CLI does not need to check the current process id for each function call.

SQL_ATTR_SYNC_POINT

A 32-bit integer value that allows the application to choose between one-phase coordinated transactions and two-phase coordinated transactions. The possible values are:

- SQL_ONEPHASE: One-phase commit is used to commit the work done by each database in a multiple database transaction. To ensure

data integrity, each transaction must not have more than one database updated. The first database that has updates performed in a transaction becomes the only updater in that transaction, all other databases accessed are treated as read-only. Any update attempts to these read-only database within this transaction are rejected.

- SQL_TWOPHASE: Two-phase commit is used to commit the work done by each database in a multiple database transaction. This requires the use of a Transaction Manager to coordinate two phase commits amongst the databases that support this protocol. Multiple readers and multiple updaters are allowed within a transaction.

Refer to the *SQL Reference* for more information on distributed unit of work (transactions).

All the connections within an application must have the same SQL_ATTR_CONNECTTYPE and SQL_ATTR_SYNC_POINT values. This attribute can also be set using the `SQLSetConnectAttr()` function. We recommend that the application set these two attributes at the environment level rather than on a per connection basis. ODBC applications written to take advantage of coordinated DB2 transactions must set these attributes at the connection level for each connection using `SQLSetConnectAttr()` as `SQLSetEnvAttr()` is not supported in ODBC.

Note: This is an IBM extension. In embedded SQL, there is an additional sync point setting called SYNCPOINT NONE. This is more restrictive than the SQL_CONCURRENT_TRANS setting of the SQL_ATTR_CONNECTTYPE attribute because SYNCPOINT NONE does not allow for multiple connections to the same database. As a result, it is not necessary for DB2 CLI to support SYNCPOINT NONE.

SQL_ATTR_USE_2BYTES_OCTET_LENGTH

A 4-byte integer value that specifies whether the SQL_DESC_OCTET_LENGTH_PTR descriptor field (used as either an APD or ARD) is a pointer to a 2 byte value or a 4 byte value (default). The possible values are:

- SQL_TRUE: SQL_DESC_OCTET_LENGTH_PTR is a pointer to a 2 byte value
- **SQL_FALSE**: SQL_DESC_OCTET_LENGTH_PTR is a pointer to a 4 byte value (default)

For more information on the descriptor field please see “SQLSetDescField - Set a Single Field of a Descriptor Record” on page 649.

This attribute can be set at any time and will take effect the next time a descriptor function is called. It does not affect the *StrLen_or_IndPtr* value for SQLGetData() calls.

Note: This is an IBM defined extension.

SQL_ATTR_USE_LIGHT_OUTPUT_SQLDA

A 32-bit integer value that specifies whether the column name is sent over the network and returned on calls to SQLDescribeCol(), SQLColAttribute(), and SQLGetDescField(). The possible values are:

- SQL_TRUE: Column name is not included in the network flow. Only the column number is returned.
- **SQL_FALSE**: Column name is included in the network flow (default).

Note: This is an IBM defined extension.

SQL_CONNECTTYPE
This *Attribute* has been replaced with SQL_ATTR_CONNECTTYPE.

SQL_MAXCONN
This *Attribute* has been replaced with SQL_ATTR_MAXCONN.

SQL_SYNC_POINT
This *Attribute* has been replaced with SQL_ATTR_SYNC_POINT.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 165. SQLSetEnvAttr SQLSTATEs

SQLSTATE	Description	Explanation
HY009	Invalid argument value.	Given the *fOption* value, an invalid value was specified for the argument *vParam*.
HY011	Operation invalid at this time.	Applications cannot set environment attributes while connection handles are allocated on the environment handle.
HY024	Invalid attribute value	Given the specified *Attribute* value, an invalid value was specified in **ValuePtr*.
HY090	Invalid string or buffer length	The *StringLength* argument was less than 0, but was not SQL_NTS.

Table 165. SQLSetEnvAttr SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY092	Option type out of range.	An invalid *Attribute* value was specified.
HYC00	Driver not capable.	The specified *Attribute* is not supported by DB2 CLI. Given specified *Attribute* value, the value specified for the argument *ValuePtr* is not supported.

Restrictions

None.

Example

See also "DB2 Multisite Update Sample" on page 56.

```
/* From CLI sample seteattr.c */
/* ... */
int main() {

    SQLHANDLE henv ;
    SQLRETURN rc ;

    SQLINTEGER output_nts = SQL_TRUE ;

/* ... */

    /* allocate an environment handle */
    rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    printf( "Setting Environment option SQL_ATTR_OUTOUT_NTS\n" ) ;

    rc = SQLSetEnvAttr( henv,
                        SQL_ATTR_OUTPUT_NTS,
                        ( SQLPOINTER ) output_nts,
                        SQL_FALSE
                      ) ;
    CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc ) ;

    rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    return( SQL_SUCCESS ) ;

}                                   /* end main */
```

References

- "SQLGetEnvAttr - Retrieve Current Environment Attribute Value" on page 481

SQLSetParam - Bind A Parameter Marker to a Buffer or LOB Locator

Deprecated

Note:

In ODBC version 3, SQLSetParam() has been deprecated and replaced with SQLBindParameter(); see "SQLBindParameter - Bind A Parameter Marker to a Buffer or LOB Locator" on page 247 for more information.

Although this version of DB2 CLI continues to support SQLSetParam(), we recommend that you begin using SQLBindParameter() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Equivalent Function: SQLBindParam()

The CLI function SQLBindParam() is exactly the same as the function SQLSetParam(). Both take the same number and type of arguments, behave the same, and return the same return codes.

SQLSetPos - Set the Cursor Position in a Rowset

Purpose

Specification:	DB2 CLI 5.0	ODBC 1	

SQLSetPos() sets the cursor position in a rowset.

Syntax

```
SQLRETURN   SQLSetPos       (SQLHSTMT          StatementHandle,
                             SQLUSMALLINT      RowNumber,
                             SQLUSMALLINT      Operation,
                             SQLUSMALLINT      LockType);
```

Function Arguments

Table 166. SQLSetPos Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLUSMALLINT	*RowNumber*	input	Position of the row in the rowset on which to perform the operation specified with the *Operation* argument. If *RowNumber* is 0, the operation applies to every row in the rowset. For additional information, see “RowNumber Argument” on page 692.

Table 166. SQLSetPos Arguments (continued)

Data Type	Argument	Use	Description
SQLUSMALLINT	*Operation*	input	Operation to perform: • SQL_POSITION • SQL_REFRESH • SQL_UPDATE • SQL_DELETE • SQL_ADD For additional information, see "Operation Argument" on page 693. ODBC also specifies the following operations for backwards compatibility only, which DB2 CLI also supports: • SQL_ADD While DB2 CLI does support SQL_ADD in `SQLSetPos()` calls, it is suggested that you use `SQLBulkOperations()` with the *Operation* argument set to SQL_ADD. See "SQLBulkOperations - Add, Update, Delete or Fetch a Set of Rows" on page 276 for more information.
SQLUSMALLINT	*LockType*	input	Specifies how to lock the row after performing the operation specified in the *Operation* argument. • SQL_LOCK_NO_CHANGE ODBC also specifies the following operations which DB2 CLI does not support: • SQL_LOCK_EXCLUSIVE • SQL_LOCK_UNLOCK For additional information, see "LockType Argument" on page 695.

Usage

RowNumber Argument

The *RowNumber* argument specifies the number of the row in the rowset on which to perform the operation specified by the *Operation* argument. If *RowNumber* is 0, the operation applies to every row in the rowset. *RowNumber* must be a value from 0 to the number of rows in the rowset.

Note In the C language, arrays are 0-based, while the *RowNumber* argument is 1-based. For example, to update the fifth row of the rowset, an application modifies the rowset buffers at array index 4, but specifies an *RowNumber* of 5.

All operations position the cursor on the row specified by *RowNumber*. The following operations require a cursor position:

- Positioned update and delete statements.
- Calls to `SQLGetData()`.
- Calls to `SQLSetPos()` with the SQL_DELETE, SQL_REFRESH, and SQL_UPDATE options.

An application can specify a cursor position when it calls `SQLSetPos()`. Generally, it calls `SQLSetPos()` with the SQL_POSITION or SQL_REFRESH operation to position the cursor before executing a positioned update or delete statement or calling `SQLGetData()`.

Operation Argument

To determine which options are supported by a data source, an application calls `SQLGetInfo()` with one of the following information types, depending on the type of cursor:

- SQL_DYNAMIC_CURSOR_ATTRIBUTES1
- SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1
- SQL_KEYSET_CURSOR_ATTRIBUTES1
- SQL_STATIC_CURSOR_ATTRIBUTES1

SQL_POSITION

DB2 CLI positions the cursor on the row specified by *RowNumber*.

The contents of the row status array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute are ignored for the SQL_POISTION *Operation*.

SQL_REFRESH

DB2 CLI positions the cursor on the row specified by *RowNumber* and refreshes data in the rowset buffers for that row. For more information about how DB2 CLI returns data in the rowset buffers, see the descriptions of row-wise and column-wise binding in `SQLFetchScroll()`.

`SQLSetPos()` with an *Operation* of SQL_REFRESH simply updates the status and content of the rows within the current fetched rowset. This includes refreshing the bookmarks. The data in the buffers is refreshed, but not refetched, so the membership in the rowset is fixed.

A successful refresh with SQLSetPos() will not change a row status of SQL_ROW_DELETED. Deleted rows within the rowset will continue to be marked as deleted until the next fetch. The rows will disappear at the next fetch if the cursor supports packing (in which case a subsequent SQLFetch() or SQLFetchScroll() does not return deleted rows).

A successful refresh with SQLSetPos() will change a row status of SQL_ROW_ADDED to SQL_ROW_SUCCESS (if the row status array exists).

A refresh with SQLSetPos() will change a row status of SQL_ROW_UPDATED to the row's new status (if the row status array exists).

If an error occurs in a SQLSetPos() operation on a row, the row status is set to SQL_ROW_ERROR (if the row status array exists).

For a cursor opened with a SQL_ATTR_CONCURRENCY statement attribute of SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES, a refresh with SQLSetPos() will update the optimistic concurrency values used by the data source to detect that the row has changed. This occurs for each row that is refreshed.

The contents of the row status array are ignored for the *SQL_REFRESH Operation*.

SQL_UPDATE

DB2 CLI positions the cursor on the row specified by *RowNumber* and updates the underlying row of data with the values in the rowset buffers (the *TargetValuePtr* argument in SQLBindCol()). It retrieves the lengths of the data from the length/indicator buffers (the *StrLen_or_IndPtr* argument in SQLBindCol()). If the length of any column is SQL_COLUMN_IGNORE, the column is not updated. After updating the row, the corresponding element of the row status array is updated to SQL_ROW_UPDATED or SQL_ROW_SUCCESS_WITH_INFO (if the row status array exists).

The row operation array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute can be used to indicate that a row in the current rowset should be ignored during a bulk update. For more information, see "Status and Operation Arrays" on page 696.

SQL_DELETE

DB2 CLI positions the cursor on the row specified by *RowNumber* and deletes the underlying row of data. It changes the corresponding element of the row status array to SQL_ROW_DELETED. After the row has been deleted, the following are not valid for the row:

- positioned update and delete statements
- calls to SQLGetData()
- calls to SQLSetPos() with *Operation* set to anything except SQL_POSITION.

Deleted rows remain visible to static and keyset-driven cursors; however, the entry in the implementation row status array (pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute) for the deleted row is changed to SQL_ROW_DELETED.

The row operation array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute can be used to indicate that a row in the current rowset should be ignored during a bulk delete. For more information, see "Status and Operation Arrays" on page 696.

SQL_ADD

ODBC also specifies the SQL_ADD *Operation* for backwards compatibility only, which DB2 CLI also supports. It is suggested, however, that you use SQLBulkOperations() with the *Operation* argument set to SQL_ADD.

See "SQLBulkOperations - Add, Update, Delete or Fetch a Set of Rows" on page 276 for more information.

LockType Argument

The *LockType* argument provides a way for applications to control concurrency. Generally, data sources that support concurrency levels and transactions will only support the SQL_LOCK_NO_CHANGE value of the *LockType* argument.

Although the *LockType* argument is specified for a single statement, the lock accords the same privileges to all statements on the connection. In particular, a lock that is acquired by one statement on a connection can be unlocked by a different statement on the same connection.

ODBC defines the following *LockType* arguments. DB2 CLI supports SQL_LOCK_NO_CHANGE. To determine which locks are supported by a data source, an application calls SQLGetInfo() with the SQL_LOCK_TYPES information type.

Table 167. Operation Values

LockType Argument	Lock Type
SQL_LOCK_NO_CHANGE	Ensures that the row is in the same locked or unlocked state as it was before SQLSetPos() was called. This value of *LockType* allows data sources that do not support explicit row-level locking to use whatever locking is required by the current concurrency and transaction isolation levels.
SQL_LOCK_EXCLUSIVE	Not supported by DB2 CLI. Locks the row exclusively.
SQL_LOCK_UNLOCK	Not supported by DB2 CLI. Unlocks the row.

Status and Operation Arrays

The following status and operation arrays are used when calling SQLSetPos():

- The row status array (as pointed to by the SQL_DESC_ARRAY_STATUS_PTR field in the IRD and the SQL_ATTR_ROW_STATUS_ARRAY statement attribute) contains status values for each row of data in the rowset. The status values are set in this array after a call to SQLFetch(), SQLFetchScroll(), or SQLSetPos. This array is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute.
- The row operation array (as pointed to by the SQL_DESC_ARRAY_STATUS_PTR field in the ARD and the SQL_ATTR_ROW_OPERATION_ARRAY statement attribute) contains a value for each row in the rowset that indicates whether a call to SQLSetPos() for a bulk operation is ignored or performed. Each element in the array is set to either SQL_ROW_PROCEED (the default) or SQL_ROW_IGNORE. This array is pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute.

The number of elements in the status and operation arrays must equal the number of rows in the rowset (as defined by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute).

For information about the row status array, see “SQLFetch - Fetch Next Row” on page 396.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 168. SQLSetPos SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The *Operation* argument was SQL_REFRESH, and string or binary data returned for a column or columns with a data type of SQL_C_CHAR or SQL_C_BINARY resulted in the truncation of non-blank character or non-NULL binary data
01S01	Error in row.	The *RowNumber* argument was 0 and an error occurred in one or more rows while performing the operation specified with the *Operation* argument. (SQL_SUCCESS_WITH_INFO is returned if an error occurs on one or more, but not all, rows of a multirow operation, and SQL_ERROR is returned if an error occurs on a single-row operation.)
01S07	Fractional truncation.	The *Operation* argument was SQL_REFRESH, the data type of the application buffer was not SQL_C_CHAR or SQL_C_BINARY, and the data returned to application buffers for one or more columns was truncated. For numeric data types, the fractional part of the number was truncated. For time and timestamp data types, the fractional portion of the time was truncated.
07006	Invalid conversion.	The data value of a column in the result set could not be converted to the data type specified by *TargetType* in the call to `SQLBindCol()`.
07009	Invalid descriptor index.	The argument *Operation* was SQL_REFRESH or SQL_UPDATE and a column was bound with a column number greater than the number of columns in the result set.
21S02	Degrees of derived table does not match column list.	The argument *Operation* was SQL_UPDATE and no columns were updateable because all columns were either unbound, read-only, or the value in the bound length/indicator buffer was SQL_COLUMN_IGNORE.
22001	String data right truncation.	The assignment of a character or binary value to a column resulted in the truncation of non-blank (for characters) or non-null (for binary)characters or bytes.

Table 168. SQLSetPos SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22003	Numeric value out of range.	The argument *Operation* was SQL_UPDATE and the assignment of a numeric value to a column in the result set caused the whole (as opposed to fractional) part of the number to be truncated. The argument *Operation* was SQL_REFRESH, and returning the numeric value for one or more bound columns would have caused a loss of significant digits.
22007	Invalid datetime format.	The argument *Operation* was SQL_UPDATE, and the assignment of a date or timestamp value to a column in the result set caused the year, month, or day field to be out of range. The argument *Operation* was SQL_REFRESH, and returning the date or timestamp value for one or more bound columns would have caused the year, month, or day field to be out of range.
22008	Datetime field overflow.	The *Operation* argument was SQL_UPDATE, and the performance of datetime arithmetic on data being sent to a column in the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the natural rules for datetimes based on the Gregorian calendar. The *Operation* argument was SQL_REFRESH, and the performance of datetime arithmetic on data being retrieved from the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the natural rules for datetimes based on the Gregorian calendar.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.

Table 168. SQLSetPos SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY008	Operation was cancelled	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, SQLCancel() was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, SQLCancel() was called on the *StatementHandle* from a different thread in a multithread application.
HY010	Function sequence error.	The specified *StatementHandle* was not in an executed state. The function was called without first calling SQLExecDirect(), SQLExecute(), or a catalog function. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. SQLExecute(), SQLExecDirect(), or SQLSetPos() was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. A Version 2 DB2 CLI application called SQLSetPos() for a *StatementHandle* before SQLFetchScroll() was called or after SQLFetch() was called, and before SQLFreeStmt() was called with the SQL_CLOSE option.
HY011	Operation invalid at this time.	A Version 2 DB2 CLI application set the SQL_ATTR_ROW_STATUS_PTR statement attribute; then SQLSetPos() was called before SQLFetch(), SQLFetchScroll(), or SQLExtendedFetch() was called.

Table 168. SQLSetPos SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY090	Invalid string or buffer length.	The *Operation* argument was SQL_ADD, SQL_UPDATE, or SQL_UPDATE_BY_BOOKMARK, a data value was a null pointer, and the column length value was not 0, SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. The *Operation* argument was SQL_ADD, SQL_UPDATE, or SQL_UPDATE_BY_BOOKMARK, a data value was not a null pointer, and the column length value was less than 0, but not equal to SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NTS, or SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. A value in a length/indicator buffer was SQL_DATA_AT_EXEC; the SQL type was either SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a other, data-source-specific data type; and the SQL_NEED_LONG_DATA_LEN information type in `SQLGetInfo()` was "Y".
HY092	Option type out of range.	The *Operation* argument was SQL_UPDATE_BY_BOOKMARK, SQL_DELETE_BY_BOOKMARK, or SQL_REFRESH_BY_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.
HY107	Row value out of range.	The value specified for the argument *RowNumber* was greater than the number of rows in the rowset.
HY109	Invalid cursor position.	The cursor associated with the *StatementHandle* was defined as forward only, so the cursor could not be positioned within the rowset. See the description for the SQL_ATTR_CURSOR_TYPE attribute in `SQLSetStmtAttr()`. The *Operation* argument was SQL_UPDATE, SQL_DELETE, or SQL_REFRESH, and the row identified by the *RowNumber* argument had been deleted or had not be fetched. The *RowNumber* argument was 0 and the *Operation* argument was SQL_POSITION.
HYC00	Driver not capable.	DB2 CLI or the data source does not support the operation requested in the *Operation* argument or the *LockType* argument.

Table 168. SQLSetPos SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HYT00	Timeout expired	The query timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr() with an *Attribute* of SQL_ATTR_QUERY_TIMEOUT.

Restrictions

None.

Example

See the README file in the sqllib\samples\cli (or sqllib/samples/cli) subdirectory for a list of appropriate samples.

References

- "SQLBindCol - Bind a Column to an Application Variable or LOB Locator" on page 227
- "SQLCancel - Cancel Statement" on page 290
- "SQLFetchScroll - Fetch Rowset and Return Data for All Bound Columns" on page 408
- "SQLGetDescField - Get Single Field Settings of Descriptor Record" on page 458
- "SQLGetDescRec - Get Multiple Field Settings of Descriptor Record" on page 463
- "SQLSetDescField - Set a Single Field of a Descriptor Record" on page 649
- "SQLSetDescRec - Set Multiple Descriptor Fields for a Column or Parameter Data" on page 677
- "SQLSetStmtAttr - Set Options Related to a Statement" on page 702

SQLSetStmtAttr - Set Options Related to a Statement

Purpose

Specification:	DB2 CLI 5.0	ODBC 3.0	ISO CLI

SQLSetStmtAttr() sets options related to a statement. To set an option for all statements associated with a specific connection, an application can call SQLSetConnectAttr().

Syntax

```
SQLRETURN   SQLSetStmtAttr   (SQLHSTMT          StatementHandle,
                              SQLINTEGER        Attribute,
                              SQLPOINTER        ValuePtr,
                              SQLINTEGER        StringLength);
```

Function Arguments

Table 169. SQLSetStmtAttr Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLINTEGER	*Attribute*	input	Option to set, listed in “Statement Attributes” on page 705

Table 169. SQLSetStmtAttr Arguments (continued)

Data Type	Argument	Use	Description
SQLHSTMT	*ValuePtr	input	If *Attribute* is an ODBC-defined attribute and *ValuePtr* points to a character string or a binary buffer, this argument should be the length of **ValuePtr*. If *Attribute* is an ODBC-defined attribute and *ValuePtr* is an integer, *StringLength* is ignored. If *Attribute* is a DB2 CLI attribute, the application indicates the nature of the attribute by setting the *StringLength* argument. *StringLength* can have the following values: • If *ValuePtr* is a pointer to a character string, then *StringLength* is the length of the string or SQL_NTS. • If *ValuePtr* is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in *StringLength*. This places a negative value in *StringLength*. • If *ValuePtr* is a pointer to a value other than a character string or a binary string, then *StringLength* should have the value SQL_IS_POINTER. • If *ValuePtr* contains a fixed-length value, then *StringLength* is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.
SQLINTEGER	*StringLength*	input	If *ValuePtr* points to a character string or a binary buffer, this argument should be the length of **ValuePtr*. If *ValuePtr* is a pointer, but not to a string or binary buffer, then *StringLength* should have the value SQL_IS_POINTER. If *ValuePtr* is not a pointer, then *StringLength* should have the value SQL_IS_NOT_POINTER.

Usage

Statement attributes for a statement remain in effect until they are changed by another call to `SQLSetStmtAttr()` or the statement is dropped by calling `SQLFreeHandle()`. Calling `SQLFreeStmt()` with the SQL_CLOSE, SQL_UNBIND, or SQL_RESET_PARAMS options does not reset statement attributes.

Some statement attributes support substitution of a similar value if the data source does not support the value specified in **ValuePtr*. In such cases, DB2

CLI returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed). For example, if *Attribute* is SQL_ATTR_CONCURRENCY, **ValuePtr* is SQL_CONCUR_ROWVER, and the data source does not support this, DB2 CLI substitutes SQL_CONCUR_VALUES and returns SQL_SUCCESS_WITH_INFO. To determine the substituted value, an application calls SQLGetStmtAttr().

The format of information set with *ValuePtr* depends on the specified *Attribute*. SQLSetStmtAttr() accepts attribute information in one of two different formats: a null-terminated character string or a 32-bit integer value. The format of each is noted in the attribute's description. This format applies to the information returned for each attribute in SQLGetStmtAttr(). Character strings pointed to by the *ValuePtr* argument of SQLSetStmtAttr() have a length of*StringLength*.

Setting Statement Attributes by Setting Descriptors

Many statement attributes also corresponding to a header field of one or more descriptors. These attributes may be set not only by a call to SQLSetStmtAttr(), but also by a call to SQLSetDescField(). Setting these options by a call to SQLSetStmtAttr(), rather than SQLSetDescField(), has the advantage that a descriptor handle does not have to be fetched.

Note: Calling SQLSetStmtAttr() for one statement can affect other statements. This occurs when the APD or ARD associated with the statement is explicitly allocated and is also associated with other statements. Because SQLSetStmtAttr() modifies the APD or ARD, the modifications apply to all statements with which this descriptor is associated. If this is not the desired behavior, the application should dissociate this descriptor from the other statement (by calling SQLSetStmtAttr() to set the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC field to a different descriptor handle) before calling SQLSetStmtAttr() again.

When a statement attribute that is also a descriptor field is set by a call to SQLSetStmtAttr(), the corresponding field in the descriptor that is associated with the statement is also set. The field is set only for the applicable descriptors that are currently associated with the statement identified by the *StatementHandle* argument, and the attribute setting does not affect any descriptors that may be associated with that statement in the future. When a descriptor field that is also a statement attribute is set by a call to SQLSetDescField(), the corresponding statement attribute is also set.

Statement attributes determine which descriptors a statement handle is associated with. When a statement is allocated (see SQLAllocHandle()), four descriptor handles are automatically allocated and associated with the

statement. Explicitly allocated descriptor handles can be associated with the statement by calling SQLAllocHandle() with an fHandleType of SQL_HANDLE_DESC to allocate a descriptor handle, then calling SQLSetStmtAttr() to associate the descriptor handle with the statement.

The following statement attributes correspond to descriptor header fields:

Table 170. Statement Attributes

Statement Attribute	Header Field	Desc.
SQL_ATTR_PARAM_BIND_OFFSET_PTR	SQL_DESC_BIND_OFFSET_PTR	APD
SQL_ATTR_PARAM_BIND_TYPE	SQL_DESC_BIND_TYPE	APD
SQL_ATTR_PARAM_OPERATION_PTR	SQL_DESC_ARRAY_STATUS_PTR	APD
SQL_ATTR_PARAM_STATUS_PTR	SQL_DESC_ARRAY_STATUS_PTR	IPD
SQL_ATTR_PARAMS_PROCESSED_PTR	SQL_DESC_ROWS_PROCESSED_PTR	IPD
SQL_ATTR_PARAMSET_SIZE	SQL_DESC_ARRAY_SIZE	APD
SQL_ATTR_ROW_ARRAY_SIZE	SQL_DESC_ARRAY_SIZE	APD
SQL_ATTR_ROW_BIND_OFFSET_PTR	SQL_DESC_BIND_OFFSET_PTR	ARD
SQL_ATTR_ROW_BIND_TYPE	SQL_DESC_BIND_TYPE	ARD
SQL_ATTR_ROW_OPERATION_PTR	SQL_DESC_ARRAY_STATUS_PTR	APD
SQL_ATTR_ROW_STATUS_PTR	SQL_DESC_ARRAY_STATUS_PTR	IRD
SQL_ATTR_ROWS_FETCHED_PTR	SQL_DESC_ROWS_PROCESSED_PTR	IRD

Statement Attributes

The currently defined attributes and the version of DB2 CLI in which they were introduced are shown below; it is expected that more will be defined to take advantage of different data sources.

Note: All statement attributes from DB2 CLI version 2 have been renamed. In version 2 they began with SQL_ but now begin with SQL_ATTR_

SQL_ATTR_APP_PARAM_DESC (DB2 CLI v5)

The handle to the APD for subsequent call to SQLExecute() and SQLExecDirect() on the statement handle. The initial value of this attribute is the descriptor implicitly allocated when the statement was initially allocated. If this value of this attribute is set to SQL_NULL_DESC, an explicitly allocated APD handle that was previously associated with the statement handle is dissociated from it, and the statement handle reverts to the implicitly allocated APD handle.

This attribute cannot be set to a descriptor handle that was implicitly allocated for another statement or to another descriptor handle that was implicitly set on the same statement; implicitly allocated descriptor handles cannot be associated with more than one statement or descriptor handle.

This attribute cannot be set at the connection level.

SQL_ATTR_APP_ROW_DESC (DB2 CLI v5)

The handle to the ARD for subsequent fetches on the statement handle. The initial value of this attribute is the descriptor implicitly allocated when the statement was initially allocated. If this value of this attribute is set to SQL_NULL_DESC, an explicitly allocated ARD handle that was previously associated with the statement handle is dissociated from it, and the statement handle reverts to the implicitly allocated ARD handle.

This attribute cannot be set to a descriptor handle that was implicitly allocated for another statement or to another descriptor handle that was implicitly set on the same statement; implicitly allocated descriptor handles cannot be associated with more than one statement or descriptor handle.

This attribute cannot be set at the connection level.

SQL_ATTR_ASYNC_ENABLE (DB2 CLI v2)

A 32-bit integer value that specifies whether a function called with the specified statement is executed asynchronously:

- **SQL_ASYNC_ENABLE_OFF** = Off (the default)
- SQL_ASYNC_ENABLE_ON = On

Once a function has been called asynchronously, only the original function, `SQLAllocHandle()`, `SQLCancel()`, `SQLSetStmtAttr()`, `SQLGetDiagField()`, `SQLGetDiagRec()`, or `SQLGetFunctions()` can be called on the statement or the connection associated with the statement, until the original function returns a code other than SQL_STILL_EXECUTING. Any other function called on the statement or the connection associated with the statement returns SQL_ERROR with an SQLSTATE of HY010 (Function sequence error). Functions can be called on other statements.

Because DB2 CLI supports statement level asynchronous-execution, the statement attribute SQL_ATTR_ASYNC_ENABLE may be set. Its initial value is the same as the value of the connection level attribute with the same name at the time the statement handle was allocated.

In general, applications should execute functions asynchronously only on single-threaded operating systems. On multi-threaded operating systems, applications should execute functions on separate threads, rather than executing them asynchronously on the same thread. DB2 CLI applications that only operate on multi-threaded operating systems do not need to support asynchronous execution. See “Writing

Multi-Threaded Applications" on page 46 and "Asynchronous Execution of CLI" on page 138 for more information.

The following functions can be executed asynchronously:

```
SQLColAttribute()        SQLGetTypeInfo()
SQLColumnPrivileges()    SQLMoreResults()
SQLColumns()             SQLNumParams()
SQLCopyDesc()            SQLNumResultCols()
SQLDescribeCol()         SQLParamData()
SQLDescribeParam()       SQLPrepare()
SQLExecDirect()          SQLPrimaryKeys()
SQLExecute()             SQLProcedureColumns()
SQLFetch()               SQLProcedures()
SQLFetchScroll()         SQLPutData()
SQLForeignKeys()         SQLSetPos()
SQLGetData()             SQLSpecialColumns()
SQLGetDescField() 1*     SQLStatistics()
SQLGetDescRec() 1*       SQLTablePrivileges()
SQLGetDiagField()        SQLTables()
SQLGetDiagRec()
```

1* These functions can be called asynchronously only if the descriptor is an implementation descriptor, not an application descriptor.

Asynchronous executing can also be set using the ASYNCENABLE DB2 CLI/ODBC configuration keyword. See "Configuring db2cli.ini" on page 159 for more information.

SQL_ATTR_BIND_TYPE (DB2 CLI v2)

A 32-bit integer value that sets the binding orientation to be used when SQLExtendedFetch() is called with this statement handle. *Column-wise binding* is selected by supplying the value **SQL_BIND_BY_COLUMN** for the argument *vParam*. *Row-wise binding* is selected by supplying a value for *vParam* specifying the length of the structure or an instance of a buffer into which result columns will be bound.

For row-wise binding, the length specified in *vParam* must include space for all of the bound columns and any padding of the structure or buffer to ensure that when the address of a bound column is incremented with the specified length, the result will point to the beginning of the same column in the next row. (When using the sizeof operator with structures or unions in ANSI C, this behavior is guaranteed.)

Column-wise binding is the default for this option.

SQL_ATTR_CLOSEOPEN (DB2 CLI v6)

To reduce the time it takes to open and close cursors, DB2 will automatically close an open cursor if a second cursor is opened using the same handle. Network flow is therefore reduced when the close request is chained with the open request and the two statements are combined into one network request (instead of two).

- **0** = DB2 acts as a regular ODBC data source: Do not chain the close and open statements, return an error if the cursor is open. This is the default.
- 1 = Chain the close and open statements.

Previous CLI applications will not benefit from this default because they are designed to explicitly close the cursor. New applications, however, can take advantage of this behavior by not closing the cursors explicitly, but by allowing CLI to close the cursor on subsequent open requests.

SQL_ATTR_CONCURRENCY (DB2 CLI v2)

A 32-bit integer value that specifies the cursor concurrency:

- SQL_CONCUR_READ_ONLY = Cursor is read-only. No updates are allowed. Supported by forward-only, static and keyset cursors.
- SQL_CONCUR_LOCK = Cursor uses the lowest level of locking sufficient to ensure that the row can be updated. Supported by forward-only and keyset cursors.
- SQL_CONCUR_VALUES = Cursor uses optimistic concurrency control, comparing values.

The default value for SQL_ATTR_CONCURRENCY is SQL_CONCUR_READ_ONLY for static and forward-only cursors. The default for a keyset cursor is SQL_CONCUR_VALUES.

This attribute can also be set through the Concurrency argument in `SQLSetScrollOptions()`. This attribute cannot be specified for an open cursor.

If the SQL_ATTR_CURSOR_TYPE *Attribute* is changed to a type that does not support the current value of SQL_ATTR_CONCURRENCY, the value of SQL_ATTR_CONCURRENCY will be changed at execution time, and a warning issued when `SQLExecDirect()` or `SQLPrepare()` is called.

If a SELECT FOR UPDATE statement is executed while the value of SQL_ATTR_CONCURRENCY is set to SQL_CONCUR_READ_ONLY, an error will be returned. If the value of SQL_ATTR_CONCURRENCY is changed to a value that is supported for some value of SQL_ATTR_CURSOR_TYPE, but not for the current value of

SQL_ATTR_CURSOR_TYPE, the value of SQL_ATTR_CURSOR_TYPE will be changed at execution time, and SQLSTATE 01S02 (Option value changed) is issued when SQLExecDirect() or SQLPrepare() is called.

If the specified concurrency is not supported by the data source, then DB2 CLI substitutes a different concurrency and returns SQLSTATE 01S02 (Option value changed). The order of substitution depends on the cursor type:

- Forward-Only: SQL_CONCUR_LOCK is substituted for SQL_CONCUR_ROWVER and SQL_CONCUR_VALUES
- Static: only SQL_CONCUR_READ_ONLY is valid
- Keyset: SQL_CONCUR_VALUES is substituted for SQL_CONCUR_ROWVER

Note: The following value has also been defined by ODBC, but is not supported by DB2 CLI

- SQL_CONCUR_ROWVER = Cursor uses optimistic concurrency control.

SQL_ATTR_CURSOR_HOLD (DB2 CLI v2)

A 32-bit integer which specifies whether the cursor associated with this *StatementHandle* is preserved in the same position as before the COMMIT operation, and whether the application can fetch without executing the statement again.

- **SQL_CURSOR_HOLD_ON** (this is the default)
- SQL_CURSOR_HOLD_OFF

The default value when an *StatementHandle* is first allocated is SQL_CURSOR_HOLD_ON.

This option cannot be specified while there is an open cursor on this *StatementHandle*.

Cursor hold can also be set using the CURSORHOLD DB2 CLI/ODBC configuration keyword. See “Configuring db2cli.ini” on page 159 for more information.

Note: This option is an IBM extension.

SQL_ATTR_CURSOR_SCROLLABLE (DB2 CLI v6)

A 32-bit integer that specifies the level of support that the application requires. Setting this attribute affects subsequent calls to SQLExecDirect() and SQLExecute(). The supported values are:

- **SQL_NONSCROLLABLE** = Scrollable cursors are not required on the statement handle. If the application calls SQLFetchScroll() on this handle, the only valid value of *FetchOrientation()* is SQL_FETCH_NEXT. This is the default.
- SQL_SCROLLABLE = Scrollable cursors are required on the statement handle. When calling SQLFetchScroll(), the application may specify any valid value of *FetchOrientation*, achieving cursor positioning in modes other than the sequential mode. For more information on scrollable cursors, see "Scrollable Cursors" on page 68.

SQL_ATTR_CURSOR_SENSITIVITY (DB2 CLI v6)

A 32-bit integer that specifies whether cursors on the statement handle make visible the changes made to a result set by another cursor. Setting this attribute affects subsequent calls to SQLExecDirect() and SQLExecute(). The supported values are:

- **SQL_UNSPECIFIED** = It is unspecified what the cursor type is and whether cursors on the statement handle make visible the changes made to a result set by another cursor. Cursors on the statement handle may make visible none, some or all such changes. This is the default.
- SQL_INSENSITIVE = All cursors on the statement handle show the result set without reflecting any changes made to it by any other cursor. Insensitive cursors are read-only. This corresponds to a static cursor which has a concurrency that is read-only.
- SQL_SENSITIVE = All cursors on the statement handle make visible all changes made to a result by another cursor.

SQL_ATTR_CURSOR_TYPE (DB2 CLI v2)

A 32-bit integer value that specifies the cursor type. The supported values are:

- **SQL_CURSOR_FORWARD_ONLY** = The cursor only scrolls forward.
- SQL_CURSOR_STATIC = The data in the result set is static. This is the default.
- SQL_CURSOR_KEYSET_DRIVEN = DB2 CLI supports a pure keyset cursor. The SQL_KEYSET_SIZE statement attribute is ignored. To limit the size of the keyset the application must limit the size of the result set by setting the SQL_ATTR_MAX_ROWS attribute to a value other than 0.

This option cannot be specified for an open cursor.

If the specified cursor type is not supported by the data source, CLI substitutes a different cursor type and returns SQLSTATE 01S02 (Option value changed). For a mixed or dynamic cursor, CLI substitutes, in order, a keyset-driven or static cursor.

Note: The following value has also been defined by ODBC, but is not supported by DB2 CLI:

- SQL_CURSOR_DYNAMIC

If this values is used, DB2 CLI sets the statement attribute to SQL_CURSOR_STATIC SQL_CURSOR_FORWARD_ONLY and returns SQLSTATE 01S02 (Option value changed). In this case the application should call `SQLGetStmtAttr()` to query the actual value.

SQL_ATTR_DEFERRED_PREPARE (DB2 CLI v5)

Specifies whether the PREPARE request is deferred until the corresponding execute request is issued.

- SQL_DEFERRED_PREPARE_OFF = Disable deferred prepare. The PREPARE request will be executed the moment it is issued.
- **SQL_DEFERRED_PREPARE_ON** (default) = Enable deferred prepare. Defer the execution of the PREPARE request until the corresponding execute request is issued. The two requests are then combined into one command/reply flow (instead of two) to minimize network flow and to improve performance.

  If the target DB2 database or the DDCS gateway does not support deferred prepare, the client disables deferred prepare for that connection.

The default behavior has changed from DB2 version 2. Deferred prepare is now the default and must be explicitly turned off if required.

Note: When deferred prepare is enabled, the row and cost estimates normally returned in the SQLERRD(3) and SQLERRD(4) of the SQLCA of a PREPARE statement may become zeros. This may be of concern to users who want to use these values to decide whether or not to continue the SQL statement.

This option is turned off if the CLI/ODBC option DB2ESTIMATE is set to a value other than zero.

Deferred prepare can also be set using the DEFERREDPREPARE DB2 CLI/ODBC configuration keyword. See “Configuring db2cli.ini” on page 159 for more information.

Note: This is an IBM defined extension.

SQL_ATTR_EARLYCLOSE (DB2 CLI v5)

Specifies whether or not the temporary cursor on the server can be automatically closed, without closing the cursor on the client, when the last record is sent to the client.

- SQL_EARLYCLOSE_OFF = Do not close the temporary cursor on the server early.
- **SQL_EARLYCLOSE_ON** = Close the temporary cursor on the server early (default).

  This saves the CLI/ODBC driver a network request by not issuing the statement to explicitly close the cursor because it knows that it has already been closed.

  Having this option on will speed up applications that make use of many small result sets.

  The EARLYCLOSE feature is not used if either:

  – The statement disqualifies for blocking.

  – The cursor type is anything other than SQL_CURSOR_FORWARD_ONLY.

The early close feature can also be set using the EARLYCLOSE DB2 CLI/ODBC configuration keyword. See “Configuring db2cli.ini” on page 159 for more information.

Note: This is an IBM defined extension.

SQL_ATTR_ENABLE_AUTO_IPD (DB2 CLI v5)

A 32-bit integer value that specifies whether automatic population of the IPD is performed:

- SQL_TRUE = Turns on automatic population of the IPD after a call to `SQLPrepare()`.
- SQL_FALSE = Turns off automatic population of the IPD after a call to `SQLPrepare()`.

The default value of the statement attribute SQL_ATTR_ENABLE_AUTO_IPD is equal to the value of the connection attribute SQL_ATTR_AUTO_IPD.

If the connection attribute SQL_ATTR_ AUTO_IPD is SQL_FALSE, the statement attribute SQL_ATTR_ENABLE_AUTO_IPD cannot be set to SQL_TRUE.

SQL_ATTR_FETCH_BOOKMARK_PTR (DB2 CLI v5)

A pointer that points to a binary bookmark value. When SQLFetchScroll() is called with *fFetchOrientation* equal to SQL_FETCH_BOOKMARK, DB2 CLI picks up the bookmark value from this field. This field defaults to a null pointer.

SQL_ATTR_IMP_PARAM_DESC (DB2 CLI v5)

The handle to the IPD. The value of this attribute is the descriptor allocated when the statement was initially allocated. The application cannot set this attribute.

This attribute can be retrieved by a call to SQLGetStmtAttr(), but not set by a call to SQLSetStmtAttr().

SQL_ATTR_IMP_ROW_DESC (DB2 CLI v5)

The handle to the IRD. The value of this attribute is the descriptor allocated when the statement was initially allocated. The application cannot set this attribute.

This attribute can be retrieved by a call to SQLGetStmtAttr(), but not set by a call to SQLSetStmtAttr().

SQL_ATTR_KEYSET_SIZE (DB2 CLI v5)

DB2 CLI supports a pure keyset cursor, therefore the SQL_KEYSET_SIZE statement attribute is ignored. To limit the size of the keyset the application must limit the size of the result set by setting the SQL_ATTR_MAX_ROWS attribute to a value other than 0.

SQL_ATTR_MAX_LENGTH (DB2 CLI v2)

A 32-bit integer value corresponding to the maximum amount of data that can be retrieved from a single character or binary column. If data is truncated because the value specified for SQL_MAX_LENGTH is less than the amount of data available, a SQLGetData() call or fetch will return SQL_SUCCESS instead of returning SQL_SUCCESS_WITH_INFO and SQLSTATE **01**004 (Data Truncated). The default value for *vParam* is **0**; 0 means that DB2 CLI will attempt to return all available data for character or binary type data.

SQL_ATTR_MAX_ROWS (DB2 CLI v2)

A 32-bit integer value corresponding to the maximum number of rows to return to the application from a query. The default value for *vParam* is **0**; 0 means all rows are returned.

SQL_ATTR_METADATA_ID (DB2 CLI v5)

A 32-bit integer value that determines how the string arguments of catalog functions are treated.

- SQL_TRUE, the string argument of catalog functions are treated as identifiers. The case if not significant. For non-delimited strings,

DB2 CLI removes any trailing spaces, and the string is folded to upper case. For delimited strings, DB2 CLI removes any leading or trailing spaces, and takes whatever is between the delimiters literally. If one of these arguments is set to a null pointer, the function returns SQL_ERROR and SQLSTATE HY009 (Invalid use of null pointer).

- **SQL_FALSE**, the string arguments of catalog functions are not treated as identifiers. The case is significant. They can either contain a string search pattern or not, depending on the argument.

  This is the default value.

The *TableType* argument of `SQLTables()`, which takes a list of values, is not affected by this attribute.

SQL_ATTR_NODESCRIBE (DB2 CLI v2)

This statement attribute is no longer required for DB2 CLI version 5 and later. Now that DB2 CLI uses deferred prepare by default, there is no need for the functionality of `SQLSetColAttributes()`. See "Deferred Prepare now on by Default" on page 773 for more details.

A 32-bit integer which specifies whether DB2 CLI should automatically describe the column attributes of the result set or wait to be informed by the application via `SQLSetColAttributes()`.

Note: This is an IBM defined extension.

SQL_ATTR_NOSCAN (DB2 CLI v2)

A 32-bit integer value that specified whether DB2 CLI will scan SQL strings for escape clauses. The two permitted values are:

- **SQL_NOSCAN_OFF** - SQL strings are scanned for escape clause sequences. This is the default.
- SQL_NOSCAN_ON - SQL strings are not scanned for escape clauses. Everything is sent directly to the server for processing.

This application can choose to turn off the scanning if it never uses vendor escape sequences in the SQL strings that it sends. This will eliminate some of the overhead processing associated with scanning.

SQL_ATTR_OPTIMIZE_FOR_NROWS (DB2 CLI v6)

A 32-bit integer value. It it is set to an integer larger than 0, ″OPTIMIZE FOR n ROWS″ clause will be appended to every select statement If set to 0 (the default) this clause will not be appended.

For more information on the effect of the OPTIMIZE FOR n ROWS clause, refer to the *Administration Guide*.

This value can also be set using the OPTIMIZEFORNROWS DB2 CLI/ODBC configuration keyword. See "Configuring db2cli.ini" on page 159 for more information.

SQL_ATTR_OPTIMIZESQLCOLUMNS (DB2 CLI v6)

A 32-bit integer.

- If set to 1, then all calls to `SQLColumns()` will be optimized if an explicit (no wildcard specified) Schema Name, explicit Table Name, and % (ALL columns) for Column Name are specified. The DB2 CLI/ODBC Driver will optimize this call so that the system tables will not be scanned.

  If the call is optimized then the COLUMN_DEF information from `SQLColumns()` (which contains the default string for the columns) is not returned, and the datatype of the AS/400 NUMERIC column will be returned as SQL_DECIMAL.
- If set to 0, information will be returned as usual. Use this setting if the application needs the COLUMN_DEF information.

This value can also be set using the OPTIMIZESQLCOLUMNS DB2 CLI/ODBC configuration keyword. See "Configuring db2cli.ini" on page 159 for more information.

SQL_ATTR_PARAM_BIND_OFFSET_PTR (DB2 CLI v5)

A 32-bit integer * value that points to an offset added to pointers to change binding of dynamic parameters. If this field is non-null, DB2 CLI dereferences the pointer, adds the dereferenced value to each of the deferred fields in the descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR), and uses the new pointer values when binding. It is set to null by default.

The bind offset is always added directly to the SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR fields. If the offset is changed to a different value, the new value is still added directly to the value in the descriptor field. The new offset is not added to the field value plus any earlier offsets.

Setting this statement attribute sets the SQL_DESC_BIND_OFFSET_PTR field in the APD header.

SQL_ATTR_PARAM_BIND_TYPE (DB2 CLI v5)

A 32-bit integer value that indicates the binding orientation to be used for dynamic parameters.

This field is set to **SQL_PARAMETER_BIND_BY_COLUMN** (the default) to select column-wise binding.

To select row-wise binding, this field is set to the length of the structure or an instance of a buffer that will be bound to a set of dynamic parameters. This length must include space for all of the bound parameters and any padding of the structure or buffer to ensure that when the address of a bound parameter is incremented with the specified length, the result will point to the beginning of the same parameter in the next set of parameters. When using the sizeof operator in ANSI C, this behavior is guaranteed.

Setting this statement attribute sets the SQL_DESC_ BIND_TYPE field in the APD header.

SQL_ATTR_PARAM_OPERATION_PTR (DB2 CLI v5)

A 16-bit unsigned integer * value that points to an array of 16-bit unsigned integer values used to ignore a parameter during execution of a SQL statement. Each value is set to either SQL_PARAM_PROCEED (for the parameter to be executed) or SQL_PARAM_IGNORE (for the parameter to be ignored).

A set of parameters can be ignored during processing by setting the status value in the array pointed to by SQL_DESC_ARRAY_STATUS_PTR in the APD to SQL_PARAM_IGNORE. A set of parameters is processed if its status value is set to SQL_PARAM_PROCEED, or if no elements in the array are set.

This statement attribute can be set to a null pointer, in which case DB2 CLI does not return parameter status values. This attribute can be set at any time, but the new value is not used until the next time `SQLExecDirect()` or `SQLExecute()` is called.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the APD.

SQL_ATTR_PARAM_STATUS_PTR (DB2 CLI v5)

A 16-bit unsigned integer * value that points to an array of UWORD values containing status information for each row of parameter values after a call to `SQLExecute()` or `SQLExecDirect()`. This field is required only if PARAMSET_SIZE is greater than 1.

The status values can contain the following values:

- SQL_PARAM_SUCCESS: The SQL statement was successfully executed for this set of parameters.

- SQL_PARAM_SUCCESS_WITH_INFO: The SQL statement was successfully executed for this set of parameters; however, warning information is available in the diagnostics data structure.
- SQL_PARAM_ERROR: There was an error in processing this set of parameters. Additional error information is available in the diagnostics data structure.
- SQL_PARAM_UNUSED: This parameter set was unused, possibly due to the fact that some previous parameter set caused an error that aborted further processing.
- SQL_PARAM_DIAG_UNAVAILABLE: DB2 CLI treats arrays of parameters as a monolithic unit and so does not generate this level of error information.

This statement attribute can be set to a null pointer, in which case DB2 CLI does not return parameter status values. This attribute can be set at any time, but the new value is not used until the next time `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()` is called.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the IPD header.

SQL_ATTR_PARAMOPT_ATOMIC (DB2 CLI v2)

This is a 32-bit integer value which determines, when `SQLParamOptions()` has been used to specify multiple values for parameter markers, whether the underlying processing should be done via ATOMIC or NOT-ATOMIC Compound SQL. The possible values are:

- **SQL_ATOMIC_YES** - The underlying processing makes use of ATOMIC Compound SQL. This is the default.
- SQL_ATOMIC_NO - The underlying processing makes use of NON-ATOMIC Compound SQL.

ATOMIC Compound SQL is not possible with: DB2 for common server prior to Version 2.1 or DRDA servers. Specifying SQL_ATOMIC_YES when connected to one of the above servers results in an error (SQLSTATE is **S1**C00).

SQL_ATTR_PARAMS_PROCESSED_PTR (DB2 CLI v5)

A 32-bit unsigned integer * record field that points to a buffer in which to return the current row number. As each row of parameters is processed, this is set to the number of that row. No row number will be returned if this is a null pointer.

Setting this statement attribute sets the SQL_DESC_ROWS_PROCESSED_PTR field in the IPD header.

SQL_ATTR_PARAMSET_SIZE (DB2 CLI v5)

A 32-bit unsigned integer value that specifies the number of values for each parameter. If SQL_ATTR_PARAMSET_SIZE is greater than 1, SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR of the APD point to arrays. The cardinality of each array is equal to the value of this field.

Setting this statement attribute sets the SQL_DESC_ARRAY_SIZE field in the APD header.

SQL_ATTR_PREFETCH (DB2 CLI v6)

This is a 32-bit value which determines if the server will prefetch the next block of data immediately after sending the current block (if supported by the server). This allows the server to get the next block of data while the application is receiving the current block.

This has no effect if the entire result set fits in the first block of data, or if the cursor is a non-blocking cursor (if it is a FOR UPDATE cursor, or if the result contains lob data, for example).

The possible values are:

- SQL_PREFETCH_ON - Prefetch will occur if supported by the server
- SQL_PREFETCH_OFF - Prefetch will not occur. This is the default.

SQL_ATTR_QUERY_OPTIMIZATION_LEVEL (DB2 V6)

A 32-bit integer value that sets the query optimization level to be used on the next call to `SQLPrepare()`, `SQLExtendedPrepare()`, or `SQLExecDirect()`.

Current values used are: 0,1,2,3,5,7, and 9. For more information about query optimization levels see the SET CURRENT QUERY OPTIMIZATION command in the *SQL Reference*.

SQL_ATTR_QUERY_TIMEOUT (DB2 CLI v2)

A 32-bit integer value that is the number of seconds to wait for an SQL statement to execute between returning to the application. DB2 CLI only supports the value of **0** except on Windows 3.1; 0 means there is no time out.

Note: On Windows 3.1, this option can be set and used to terminate long running queries. If this is specified, the underlying Windows 3.1 connectivity code will display a dialog box to inform the user that the specified number of seconds have elapsed and prompt the user to continue or interrupt the query.

This option is not valid for platforms other than Windows 3.1, **S1**C00 is returned.

SQL_ATTR_RETRIEVE_DATA (DB2 CLI v2)

A 32-bit integer value:

- **SQL_RD_ON** = `SQLFetchScroll()` and in DB2 CLI Version 5.2 and later, `SQLFetch()`, retrieve data after it positions the cursor to the specified location. This is the default.
- SQL_RD_OFF = `SQLFetchScroll()` and in DB2 CLI Version 5.2 and later, `SQLFetch()`, do not retrieve data after it positions the cursor.

By setting SQL_RETRIEVE_DATA to SQL_RD_OFF, an application can verify if a row exists or retrieve a bookmark for the row without incurring the overhead of retrieving rows.

SQL_ATTR_ROW_ARRAY_SIZE (DB2 CLI v5)

A 32-bit integer value that specifies the number of rows in the rowset. This is the number of rows returned by each call to `SQLFetch()` or `SQLFetchScroll()`. The default value is 1.

If the specified rowset size exceeds the maximum rowset size supported by the data source, DB2 CLI substitutes that value and returns SQLSTATE 01S02 (Option value changed).

This option can be specified for an open cursor and can also be set through the RowsetSize argument in `SQLSetScrollOptions()`.

Setting this statement attribute sets the SQL_DESC_ARRAY_SIZE field in the ARD header.

SQL_ATTR_ROW_BIND_OFFSET_PTR (DB2 CLI v5)

A 32-bit integer * value that points to an offset added to pointers to change binding of column data. If this field is non-null, DB2 CLI dereferences the pointer, adds the dereferenced value to each of the deferred fields in the descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR), and uses the new pointer values when binding. It is set to null by default.

Setting this statement attribute sets the SQL_DESC_BIND_OFFSET_PTR field in the ARD header.

SQL_ATTR_ROW_BIND_TYPE (DB2 CLI v5)

A 32-bit integer value that sets the binding orientation to be used when `SQLFetch()` or `SQLFetchScroll()` is called on the associated statement. Column-wise binding is selected by supplying the defined constant SQL_BIND_BY_COLUMN in **ValuePtr*. Row-wise binding is selected by supplying a value in **ValuePtr* specifying the length of a structure or an instance of a buffer into which result columns will be bound.

The length specified in **ValuePtr* must include space for all of the bound columns and any padding of the structure or buffer to ensure that when the address of a bound column is incremented with the specified length, the result will point to the beginning of the same column in the next row. When using the **sizeof** operator with structures or unions in ANSI C, this behavior is guaranteed.

Column-wise binding is the default binding orientation for SQLFetch() and SQLFetchScroll().

Setting this statement attribute sets the SQL_DESC_BIND_TYPE field in the ARD header.

SQL_ATTR_ROW_NUMBER (DB2 CLI v5)

A 32-bit integer value that is the number of the current row in the entire result set. If the number of the current row cannot be determined or there is no current row, DB2 CLI returns 0.

This attribute can be retrieved by a call to SQLGetStmtAttr(), but not set by a call to SQLSetStmtAttr().

SQL_ATTR_ROW_OPERATION_PTR (DB2 CLI v5)

A 16-bit unsigned integer * value that points to an array of UDWORD values used to ignore a row during a bulk operation using SQLSetPos(). Each value is set to either SQL_ROW_PROCEED (for the row to be included in the bulk operation) or SQL_ROW_IGNORE (for the row to be excluded from the bulk operation).

This statement attribute can be set to a null pointer, in which case DB2 CLI does not return row status values. This attribute can be set at any time, but the new value is not used until the next time SQLFetch(), SQLFetchScroll(), or SQLSetPos() is called.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the ARD.

SQL_ATTR_ROW_STATUS_PTR (DB2 CLI v5)

A 16-bit unsigned integer * value that points to an array of UWORD values containing row status values after a call to SQLFetch() or SQLFetchScroll(). The array has as many elements as there are rows in the rowset.

This statement attribute can be set to a null pointer, in which case DB2 CLI does not return row status values. This attribute can be set at any time, but the new value is not used until the next time SQLFetch(), SQLFetchScroll(), or SQLSetPos() is called.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the IRD header.

SQL_ATTR_ROWS_FETCHED_PTR (DB2 CLI v5)

A 32-bit unsigned integer * value that points to a buffer in which to return the number of rows fetched after a call to SQLFetch() or SQLFetchScroll().

Setting this statement attribute sets the SQL_DESC_ROWS_PROCESSED_PTR field in the IRD header.

This attribute is mapped by DB2 CLI to the RowCountPtr array in a call to SQLExtendedFetch().

SQL_ATTR_ROWSET_SIZE (DB2 CLI v2)

DB2 CLI applications should now use SQLFetchScroll() rather than SQLExtendedFetch(). Applications should also use the statement attribute SQL_ATTR_ROW_ARRAY_SIZE to set the number of rows in the rowset. See "Specifying the Rowset Returned from the Result Set" on page 71 for more information.

A 32-bit integer value that specifies the number of rows in the rowset. A rowset is the array of rows returned by each call to SQLExtendedFetch(). The default value is **1**, which is equivalent to making a single SQLFetch(). This option can be specified even when the cursor is open and becomes effective on the next SQLExtendedFetch() call.

SQL_ATTR_SIMULATE_CURSOR (DB2 CLI v5)

This statement attribute is not supported by DB2 CLI but is defined by ODBC.

A 32-bit integer value that specifies whether simulated positioned update and delete statements guarantee that such statements affect only one single row.

SQL_ATTR_STMTTXN_ISOLATION (DB2 CLI v2)

See SQL_ATTR_TXN_ISOLATION below.

SQL_ATTR_TXN_ISOLATION (DB2 CLI v2)

A 32-bit integer value that sets the transaction isolation level for the current *StatementHandle*.

This option cannot be set if there is an open cursor on this statement handle (SQLSTATE **24**000).

The value SQL_ATTR_STMTTXN_ISOLATION is synonymous with SQL_ATTR_TXN_ISOLATION. However, since the ODBC Driver Manager will reject the setting of SQL_ATTR_TXN_ISOLATION as a statement option, ODBC applications that need to set translation

isolation level on a per statement basis must use the manifest constant SQL_ATTR_STMTTXN_ISOLATION instead on the SQLSetStmtAttr() call.

The transaction isolation level can also be set using the TXNISOLATION DB2 CLI/ODBC configuration keyword. See "Configuring db2cli.ini" on page 159 for more information.

This attribute (or corresponding keyword) is only applicable if the default isolation level is used. If the application has specifically set the isolation level then this attribute will have no effect.

Note: It is an IBM extension to allow setting this option at the statement level.

SQL_ATTR_USE_BOOKMARKS (DB2 CLI v5)

A 32-bit integer value that specifies whether an application will use bookmarks with a cursor:

- **SQL_UB_OFF** = Off (the default)
- SQL_UB_VARIABLE = An application will use bookmarks with a cursor, and DB2 CLI will provide variable-length bookmarks if they are supported.

To use bookmarks with a cursor, the application must specify this option with the SQL_UB_VARIABLE value before opening the cursor.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 171. SQLSetStmtAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed.	DB2 CLI did not support the value specified in **ValuePtr*, or the value specified in **ValuePtr* was invalid because of SQL constraints or requirements, so DB2 CLI substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)

Table 171. SQLSetStmtAttr SQLSTATEs (continued)

SQLSTATE	Description	Explanation
08S01	Communication link failure.	The communication link between DB2 CLI and the data source to which it was connected failed before the function completed processing.
24000	Invalid cursor state.	The *Attribute* was SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, SQL_ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS and the cursor was open.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI was unable to allocate memory required to support execution or completion of the function.
HY009	Invalid argument value.	A null pointer was passed for *ValuePtr* and the value in **ValuePtr* was a string attribute.
HY010	Function sequence error.	An asynchronously executing function was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()` or `SQLExecDirect()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY011	Operation invalid at this time.	The *Attribute* was SQL_ATTR_CONCURRENCY, SQL_ ATTR_CURSOR_TYPE, SQL_ ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS and the statement was prepared.
HY017	Invalid use of an automatically allocated descriptor handle.	The *Attribute* argument was SQL_ATTR_IMP_ROW_DESC or SQL_ATTR_IMP_PARAM_DESC. The *Attribute* argument was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and the value in **ValuePtr* was an implicitly allocated descriptor handle.
HY024	Invalid attribute value.	Given the specified *Attribute* value, an invalid value was specified in **ValuePtr*. (DB2 CLI returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE or SQL_ATTR_ASYNC_ENABLE. For all other connection and statement attributes, the driver must verify the value specified in **ValuePtr*.)
HY090	Invalid string or buffer length.	The *StringLength* argument was less than 0, but was not SQL_NTS.

Table 171. SQLSetStmtAttr SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid for this version of DB2 CLI.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid connection or statement attribute for the version of the DB2 CLI driver, but was not supported by the data source.

Restrictions

None.

Example

See `SQLFetchScroll()`.

References

- "SQLCancel - Cancel Statement" on page 290
- "SQLGetConnectAttr - Get Current Attribute Setting" on page 439
- "SQLGetStmtAttr - Get Current Setting of a Statement Attribute" on page 545
- "SQLSetConnectAttr - Set Connection Attributes" on page 618
- "SQLSetDescField - Set a Single Field of a Descriptor Record" on page 649

SQLSetStmtOption - Set Statement Option

Deprecated

Note:

In ODBC version 3, SQLSetStmtOption() has been deprecated and replaced with SQLSetStmtAttr(); see "SQLSetStmtAttr - Set Options Related to a Statement" on page 702 for more information.

Although this version of DB2 CLI continues to support SQLSetStmtOption(), we recommend that you begin using SQLSetStmtAttr() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Note: This deprecated function cannot be used in a 64-bit environment. For more details see "Deprecated Functions Not Supported in a 64-bit Environment" on page 768.

Migrating to the New Function

The statement:

```
SQLSetStmtOption(
           hstmt,
           SQL_ROWSET_SIZE,
           RowSetSize);
```

for example, would be rewritten using the new function as:

```
SQLSetStmtAttr(
           hstmt,
           SQL_ATTR_ROW_ARRAY_SIZE,
           (SQLPOINTER) RowSetSize,
           0);
```

SQLSpecialColumns - Get Special (Row Identifier) Columns

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLSpecialColumns() returns unique row identifier information (primary key or unique index) for a table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLSpecialColumns(
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLUSMALLINT      IdentifierType,    /* fColType */
                SQLCHAR      FAR  *CatalogName,      /* szCatalogName */
                SQLSMALLINT       NameLength1,       /* cbCatalogName */
                SQLCHAR      FAR  *SchemaName,       /* szSchemaName */
                SQLSMALLINT       NameLength2,       /* cbSchemaName */
                SQLCHAR      FAR  *TableName,        /* szTableName */
                SQLSMALLINT       NameLength3,       /* cbTableName */
                SQLUSMALLINT      Scope,             /* fScope */
                SQLUSMALLINT      Nullable);         /* fNullable */
```

Function Arguments

Table 172. SQLSpecialColumns Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle
SQLUSMALLINT	IdentifierType	Input	Type of unique row identifier to return. Only the following type is supported: • SQL_BEST_ROWID Returns the optimal set of column(s) which can uniquely identify any row in the specified table. **Note:** For compatibility with ODBC applications, SQL_ROWVER is also recognized, but not supported; therefore, if SQL_ROWVER is specified, an empty result will be returned.
SQLCHAR *	CatalogName	Input	Catalog qualifier of a 3 part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	NameLength1	Input	Length of *CatalogName*. This must be a set to 0.
SQLCHAR *	SchemaName	Input	Schema qualifier of the specified table.
SQLSMALLINT	NameLength2	Input	Length of *SchemaName*.

Table 172. SQLSpecialColumns Arguments (continued)

Data Type	Argument	Use	Description
SQLCHAR *	TableName	Input	Table name.
SQLSMALLINT	NameLength3	Input	Length of *NameLength3*.
SQLUSMALLINT	Scope	Input	Minimum required duration for which the unique row identifier will be valid. *Scope* must be one of the following: • SQL_SCOPE_CURROW: The row identifier is guaranteed to be valid only while positioned on that row. A later re-select using the same row identifier values may not return a row if the row was updated or deleted by another transaction. • SQL_SCOPE_TRANSACTION: The row identifier is guaranteed to be valid for the duration of the current transaction. • SQL_SCOPE_SESSION: The row identifier is guaranteed to be valid for the duration of the connection. The duration over which a row identifier value is guaranteed to be valid depends on the current transaction isolation level. For information and scenarios involving isolation levels, refer to the IBM DB2 SQL Reference.
SQLUSMALLINT	Nullable	Input	Determines whether to return special columns that can have a NULL value. Must be one of the following: • SQL_NO_NULLS - The row identifier column set returned cannot have any NULL values. • SQL_NULLABLE - The row identifier column set returned may include columns where NULL values are permitted.

Usage

If multiple ways exist to uniquely identify any row in a table (i.e. if there are multiple unique indexes on the specified table), then DB2 CLI will return the *best* set of row identifier column set based on its internal criterion.

If there is no column set which allow any row in the table to be uniquely identified, an empty result set is returned.

SQLSpecialColumns

The unique row identifier information is returned in the form of a result set where each column of the row identifier is represented by one row in the result set. “Columns Returned By SQLSpecialColumns” shows the order of the columns in the result set returned by `SQLSpecialColumns()`, sorted by SCOPE.

Since calls to `SQLSpecialColumns()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the SQL_MAX_COLUMN_NAME_LEN to determine the actual length of the COLUMN_NAME column supported by the connected DBMS.

Although new columns may be added and the names of the columns changed in future releases, the position of the current columns will not change.

Columns Returned By SQLSpecialColumns

Column 1 SCOPE (SMALLINT)
The duration for which the name in COLUMN_NAME is guaranteed to point to the same row. Valid values are the same as for the *Scope* argument: Actual scope of the row identifier. Contains one of the following values:

- SQL_SCOPE_CURROW
- SQL_SCOPE_TRANSACTION
- SQL_SCOPE_SESSION

Refer to *Scope* in Table 172 on page 726 for a description of each value.

Column 2 COLUMN_NAME (VARCHAR(128) not NULL)
Name of the column that is (or part of) the table’s primary key.

Column 3 DATA_TYPE (SMALLINT not NULL)
SQL data type of the column. One of the values in the Symbolic SQL Data Type column in Table 2 on page 29.

Column 4 TYPE_NAME (VARCHAR(128) not NULL)
DBMS character string represented of the name associated with DATA_TYPE column value.

Column 5 COLUMN_SIZE (INTEGER)
If the DATA_TYPE column value denotes a character or binary string,

then this column contains the maximum length in bytes; if it is a graphic (DBCS) string, this is the number of double byte characters for the parameter.

For date, time, timestamp data types, this is the total number of bytes required to display the value when converted to character.

For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.

See also Table 194 on page 817.

Column 6 BUFFER_LENGTH (INTEGER)

The maximum number of bytes for the associated C buffer to store data from this column if SQL_C_DEFAULT were specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` calls. This length does not include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.

See also Table 196 on page 819.

Column 7 DECIMAL_DIGITS (SMALLINT)

The scale of the column. NULL is returned for data types where scale is not applicable. See also Table 195 on page 819.

Column 8 PSEUDO_COLUMN (SMALLINT)

Indicates whether or not the column is a pseudo-column DB2 Call Level Interface will only return:

- SQL_PC_NOT_PSEUDO

DB2 DBMSs do not support pseudo columns. ODBC applications may receive the following values from other non-IBM RDBMS servers:

- SQL_PC_UNKNOWN
- SQL_PC_PSEUDO

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

SQLSpecialColumns

Diagnostics

Table 173. SQLSpecialColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY009	Invalid argument value.	*TableName* is null.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the length arguments was less than 0, but not equal to SQL_NTS. The value of one of the length arguments exceeded the maximum length supported by the DBMS for that qualifier or name.
HY097	Column type out of range.	An invalid *IdentifierType* value was specified.
HY098	Scope type out of range.	An invalid *Scope* value was specified.
HY099	Nullable type out of range.	An invalid *Nullable* values was specified.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.

Table 173. SQLSpecialColumns SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

None.

Example

```
/* From CLI sample browser.c */
/* ... */
SQLRETURN list_index_columns( SQLHANDLE hstmt,
                              SQLCHAR * schema,
                              SQLCHAR * tablename
                            ) {

/* ... */
    rc = SQLSpecialColumns(hstmt, SQL_BEST_ROWID, NULL, 0, schema, SQL_NTS,
                    tablename, SQL_NTS, SQL_SCOPE_CURROW, SQL_NULLABLE);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) column_name.s, 129,
                    &column_name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) type_name.s, 129,
                    &type_name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 5, SQL_C_LONG, (SQLPOINTER) & precision,
                    sizeof(precision), &precision_ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 7, SQL_C_SHORT, (SQLPOINTER) & scale,
                    sizeof(scale), &scale_ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    printf("Primary Key or Unique Index for %s.%s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("   %s, %s ", column_name.s, type_name.s);
        if (precision_ind != SQL_NULL_DATA) {
            printf(" (%ld", precision);
        } else {
            printf("(\n");
        }
        if (scale_ind != SQL_NULL_DATA) {
```

```
            printf(", %d)\n", scale);
        } else {
            printf(")\n");
        }
    }
```

References

- "SQLColumns - Get Column Information for a Table" on page 315
- "SQLStatistics - Get Index and Statistics Information For A Base Table" on page 733
- "SQLTables - Get Table Information" on page 745

SQLStatistics - Get Index and Statistics Information For A Base Table

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLStatistics() retrieves index information for a given table. It also returns the cardinality and the number of pages associated with the table and the indexes on the table. The information is returned in a result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLStatistics    (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLCHAR      FAR *CatalogName,       /* szCatalogName */
                SQLSMALLINT       NameLength1,       /* cbCatalogName */
                SQLCHAR      FAR *SchemaName,        /* szSchemaName */
                SQLSMALLINT       NameLength2,       /* cbSchemaName */
                SQLCHAR      FAR *TableName,         /* szTableName */
                SQLSMALLINT       NameLength3,       /* cbTableName */
                SQLUSMALLINT      Unique,            /* fUnique */
                SQLUSMALLINT      Reserved);         /* fAccuracy */
```

Function Arguments

Table 174. SQLStatistics Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.
SQLCHAR *	CatalogName	Input	Catalog qualifier of a 3 part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	NameLength1	Input	Length of *NameLength1*. This must be set to 0.
SQLCHAR *	SchemaName	Input	Schema qualifier of the specified table.
SQLSMALLINT	NameLength2	Input	Length of *SchemaName*.
SQLCHAR *	TableName	Input	Table name.
SQLSMALLINT	NameLength3	Input	Length of *NameLength3*.
SQLUSMALLINT	Unique	Input	Type of index information to return: • SQL_INDEX_UNIQUE Only unique indexes will be returned. • SQL_INDEX_ALL All indexes will be returned.

Table 174. SQLStatistics Arguments (continued)

Data Type	Argument	Use	Description
SQLUSMALLINT	Reserved	Input	Indicate whether the CARDINALITY and PAGES columns in the result set contain the most current information: • SQL_ENSURE : This value is reserved for future use, when the application requests the most up to date statistics information. **New applications should not use this value**. Existing applications specifying this value will receive the same results as SQL_QUICK. • SQL_QUICK : Statistics which are readily available at the server are returned. The values may not be current, and no attempt is made to ensure that they be up to date.

Usage

SQLStatistics() returns two types of information:

- Statistics information for the table (if it is available):
  - when the TYPE column in the table below is set to SQL_TABLE_STAT, the number of rows in the table and the number of pages used to store the table.
  - when the TYPE column indicates an index, the number of unique values in the index, and the number of pages used to store the indexes.
- Information about each index, where each index column is represented by one row of the result set. The result set columns are given in "Columns Returned By SQLStatistics" on page 735 in the order shown; the rows in the result set are ordered by NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME and ORDINAL_POSITION.

Since calls to SQLStatistics() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns Returned By SQLStatistics

Column 1 TABLE_CAT (VARCHAR(128))
This is always null.

Column 2 TABLE_SCHEM (VARCHAR(128))
The name of the schema containing TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL)
Name of the table.

Column 4 NON_UNIQUE (SMALLINT)
Indicates whether the index prohibits duplicate values:

- SQL_TRUE if the index allows duplicate values.
- SQL_FALSE if the index values must be unique.
- NULL is returned if the TYPE column indicates that this row is SQL_TABLE_STAT (statistics information on the table itself).

Column 5 INDEX_QUALIFIER (VARCHAR(128))
The string that would be used to qualify the index name in the DROP INDEX statement. Appending a period (.) plus the INDEX_NAME results in a full specification of the index.

Column 6 INDEX_NAME (VARCHAR(128))
The name of the index. If the TYPE column has the value SQL_TABLE_STAT, this column has the value NULL.

Column 7 TYPE (SMALLINT not NULL)
Indicates the type of information contained in this row of the result set:

- SQL_TABLE_STAT - Indicates this row contains statistics information on the table itself.
- SQL_INDEX_CLUSTERED - Indicates this row contains information on an index, and the index type is a clustered index.
- SQL_INDEX_HASHED - Indicates this row contains information on an index, and the index type is a hashed index.
- SQL_INDEX_OTHER - Indicates this row contains information on an index, and the index type is other than clustered or hashed.

Column 8 ORDINAL_POSITION (SMALLINT)
Ordinal position of the column within the index whose name is given in the INDEX_NAME column. A NULL value is returned for this column if the TYPE column has the value of SQL_TABLE_STAT.

Column 9 COLUMN_NAME (VARCHAR(128))
Name of the column in the index. A NULL value is returned for this column if the TYPE column has the value of SQL_TABLE_STAT.

Column 10 ASC_OR_DESC (CHAR(1))
Sort sequence for the column; "**A**" for ascending, "**D**" for descending. NULL value is returned if the value in the TYPE column is SQL_TABLE_STAT.

Column 11 CARDINALITY (INTEGER)

- If the TYPE column contains the value SQL_TABLE_STAT, this column contains the number of rows in the table.
- If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of unique values in the index.
- A NULL value is returned if information is not available from the DBMS.

Column 12 PAGES (INTEGER)

- If the TYPE column contains the value SQL_TABLE_STAT, this column contains the number of pages used to store the table.
- If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of pages used to store the indexes.
- A NULL value is returned if information is not available from the DBMS.

Column 13 FILTER_CONDITION (VARCHAR(128))
If the index is a filtered index, this is the filter condition. Since DATABASE 2 servers do not support filtered indexes, NULL is always returned. NULL is also returned if TYPE is SQL_TABLE_STAT.

For the row in the result set that contains table statistics (TYPE is set to SQL_TABLE_STAT), the columns values of NON_UNIQUE, INDEX_QUALIFIER, INDEX_NAME, ORDINAL_POSITION, COLUMN_NAME, and ASC_OR_DESC are set to NULL. If the CARDINALITY or PAGES information cannot be determined, then NULL is returned for those columns.

Note: The accuracy of the information returned in the SQLERRD(3) and SQLERRD(4) fields is dependent on many factors such as the use of parameter markers and expressions within the statement. The main factor which can be controlled is the accuracy of the database statistics. That is, when the statistics were last updated, (for example, for DB2 Universal Database, the last time the `RUNSTATS` command was run).

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 175. SQLStatistics SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY009	Invalid argument value.	*TableName* is null.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.

Table 175. SQLStatistics SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS. The valid of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the SQLGetInfo() function.
HY100	Uniqueness option type out of range.	An invalid *Unique* value was specified.
HY101	Accuracy option type out of range.	An invalid *Reserved* value was specified.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

None.

Example

```
/* From CLI sample browser.c */
/* ... */
SQLRETURN list_stats( SQLHANDLE hstmt,
                      SQLCHAR * schema,
                      SQLCHAR * tablename
                    ) {

/* ... */

    rc = SQLStatistics(hstmt, NULL, 0, schema, SQL_NTS,
                    tablename, SQL_NTS, SQL_INDEX_UNIQUE, SQL_QUICK);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 4, SQL_C_SHORT,
                          &non_unique, 2, &non_unique_ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR,
                          index_name.s, 129, &index_name.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 7, SQL_C_SHORT,
```

```
                        &type, 2, &type_ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 9, SQL_C_CHAR,
                        column_name.s, 129, &column_name.ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 11, SQL_C_LONG,
                        &cardinality, 4, &card_ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 12, SQL_C_LONG,
                        &pages, 4, &pages_ind);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

printf("Statistics for %s.%s\n", schema, tablename);

while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
{  if (type != SQL_TABLE_STAT)
   {   printf("  Column: %-18s Index Name: %-18s\n",
               column_name.s, index_name.s);
   }
   else
   {   printf("  Table Statistics:\n");
   }
   if (card_ind != SQL_NULL_DATA)
      printf("    Cardinality = %13ld", cardinality);
   else
      printf("    Cardinality = (Unavailable)");

   if (pages_ind != SQL_NULL_DATA)
      printf(" Pages = %13ld\n", pages);
   else
      printf(" Pages = (Unavailable)\n");
}
```

References

- “SQLColumns - Get Column Information for a Table” on page 315
- “SQLSpecialColumns - Get Special (Row Identifier) Columns” on page 726

SQLTablePrivileges - Get Privileges Associated With A Table

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLTablePrivileges() returns a list of tables and associated privileges for each table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN SQLTablePrivileges (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLCHAR      FAR  *CatalogName,      /*  szCatalogName */
                SQLSMALLINT       NameLength1,       /* cbCatalogName */
                SQLCHAR      FAR  *SchemaName,       /* szSchemaName */
                SQLSMALLINT       NameLength2,       /* cbSchemaName */
                SQLCHAR      FAR  *TableName,        /* szTableName */
                SQLSMALLINT       NameLength3);      /* cbTableName */
```

Function Arguments

Table 176. SQLTablePrivileges Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.
SQLCHAR *	szTableQualifier	Input	Catalog qualifier of a 3 part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	cbTableQualifier	Input	Length of *CatalogName*. This must be set to 0.
SQLCHAR *	SchemaName	Input	Buffer that may contain a *pattern-value* to qualify the result set by schema name.
SQLSMALLINT	NameLength2	Input	Length of *SchemaName*.
SQLCHAR *	TableName	Input	Buffer that may contain a *pattern-value* to qualify the result set by table name.
SQLSMALLINT	NameLength3	Input	Length of *TableName*.

Note that the *SchemaName* and *TableName* arguments accept search pattern. For more information about valid search patterns, refer to “Input Arguments on Catalog Functions” on page 66.

Usage

The results are returned as a standard result set containing the columns listed in the following table. The result set is ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and PRIVILEGE. If multiple privileges are associated with any given table, each privilege is returned as a separate row.

The granularity of each privilege reported here may or may not apply at the column level; for example, for some data sources, if a table can be updated, every column in that table can also be updated. For other data sources, the application must call SQLColumnPrivileges() to discover if the individual columns have the same table privileges.

Since calls to SQLTablePrivileges() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns Returned By SQLTablePrivileges

Column 1 TABLE_CAT (VARCHAR(128))
This is always null.

Column 2 TABLE_SCHEM (VARCHAR(128))
The name of the schema contain TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL)
The name of the table.

Column 4 GRANTOR (VARCHAR(128))
Authorization ID of the user who granted the privilege.

Column 5 GRANTEE (VARCHAR(128))
Authorization ID of the user to whom the privilege is granted.

Column 6 PRIVILEGE (VARCHAR(128))
The table privilege. This may be one of the following strings:

- ALTER
- CONTROL
- INDEX
- DELETE
- INSERT

- REFERENCES
- SELECT
- UPDATE

Column 7 IS_GRANTABLE (VARCHAR(3))
Indicates whether the grantee is permitted to grant the privilege to other users.

This can be ″YES″, ″NO″ or NULL.

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLProcedures()` result set in ODBC.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 177. SQLTablePrivileges SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY009	Invalid argument value.	*TableName* is null.

Table 177. SQLTablePrivileges SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS. The valid of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the SQLGetInfo() function.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

None.

Example

```
/* From CLI sample browser.c */
/* ... */
SQLRETURN list_table_privileges( SQLHANDLE hstmt,
                                 SQLCHAR * schema,
                                 SQLCHAR * tablename
                               ) {

    SQLRETURN       rc;
    struct { SQLINTEGER ind;   /* Length & Indicator variable */
             SQLCHAR  s[129];  /* String variable */
           } grantor, grantee, privilege;

    struct { SQLINTEGER  ind;
```

```
            SQLCHAR     s[4];
           }is_grantable;

    SQLCHAR         cur_name[512] = "";  /* Used when printing the */
    SQLCHAR         pre_name[512] = "";  /* Result set */

    /* Create Table Privilges result set */
    rc = SQLTablePrivileges(hstmt, NULL, 0, schema, SQL_NTS,
                            tablename, SQL_NTS);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) grantor.s, 129,
                    &grantor.ind);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Continue Binding, then fetch and display result set */
```

References

- “SQLTables - Get Table Information” on page 745

SQLTables - Get Table Information

Purpose

Specification:	DB2 CLI 2.1	ODBC 1.0	

SQLTables() returns a list of table names and associated information stored in the system catalog of the connected data source. The list of table names is returned as a result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLTables        (
                SQLHSTMT          StatementHandle,   /* hstmt */
                SQLCHAR      FAR  *CatalogName,      /* szCatalogName */
                SQLSMALLINT       NameLength1,       /* cbCatalogName */
                SQLCHAR      FAR  *SchemaName,       /* szSchemaName */
                SQLSMALLINT       NameLength2,       /* cbSchemaName */
                SQLCHAR      FAR  *TableName,        /* szTableName */
                SQLSMALLINT       NameLength3,       /* cbTableName */
                SQLCHAR      FAR  *TableType,        /* szTableType */
                SQLSMALLINT       NameLength4);      /* cbTableType */
```

Function Arguments

Table 178. SQLTables Arguments

Data Type	Argument	Use	Description
SQLHSTMT	StatementHandle	Input	Statement handle.
SQLCHAR *	CatalogName	Input	Buffer that may contain a *pattern-value* to qualify the result set. *Catalog* is the first part of a 3 part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	NameLength1	Input	Length of *CatalogName*. This must be set to 0.
SQLCHAR *	SchemaName	Input	Buffer that may contain a *pattern-value* to qualify the result set by schema name.
SQLSMALLINT	NameLength2	Input	Length of *SchemaName*.
SQLCHAR *	TableName	Input	Buffer that may contain a *pattern-value* to qualify the result set by table name.
SQLSMALLINT	NameLength3	Input	Length of *TableName*.

Table 178. SQLTables Arguments (continued)

Data Type	Argument	Use	Description
SQLCHAR *	TableType	Input	Buffer that may contain a *value list* to qualify the result set by table type. The value list is a list of upper-case comma-separated single quoted values for the table types of interest. Valid table type identifiers may include: TABLE, VIEW, SYSTEM TABLE, ALIAS, SYNONYM. If *TableType* argument is a NULL pointer or a zero length string, then this is equivalent to specifying all of the possibilities for the table type identifier. If SYSTEM TABLE is specified, then both system tables and system views (if there are any) are returned.
SQLSMALLINT	NameLength4	Input	Size of *NameLength4*

Note that the *CatalogName*, *SchemaName*, and *TableName* arguments accept search patterns. For more information about valid search patterns, refer to "Input Arguments on Catalog Functions" on page 66.

Usage

Table information is returned in a result set where each table is represented by one row of the result set. To determine the type of access permitted on any given table in the list, the application can call `SQLTablePrivileges()`. Otherwise, the application must be able to handle a situation where the user selects a table for which SELECT privileges are not granted.

To support obtaining just a list of schemas, the following special semantics for the *SchemaName* argument can be applied: if *SchemaName* is a string containing a single percent (%) character, and *CatalogName* and *TableName* are empty strings, then the result set contains a list of valid schemas in the data source.

If *TableType* is a single percent character (%) and *CatalogName*, *SchemaName*, and *TableName* are empty strings, then the result set contains a list of valid table types for the data source. (All columns except the TABLE_TYPE column contain NULLs.)

If *TableType* is not an empty string, it must contain a list of upper-case, comma-separated values for the types of interest; each value may be enclosed in single quotes or unquoted. For example, "'TABLE','VIEW'" or

″TABLE,VIEW″. If the data source does not support or does not recognize a specified table type, nothing is returned for that type.

Sometimes, an application calls SQLTables() with null pointers for some or all of the *SchemaName, TableName,* and *TableType* arguments so that no attempt is made to restrict the result set returned. For some data sources that contain a large quantity of tables, views, aliases, etc., this scenario maps to an extremely large result set and very long retrieval times. Three mechanisms are introduced to help the end user reduce the long retrieval times: three keywords (SCHEMALIST, SYSCHEMA, TABLETYPE) can be specified in the CLI initialization file to help restrict the result set when the application has supplied null pointers for either or both of *SchemaName* and *TableType*. These keywords and their usage are discussed in detail in “Configuration Keywords” on page 164. If the application did not specify a null pointer for *SchemaName* or *TableType* then the associated keyword specification in the CLI initialization file is ignored.

The result set returned by SQLTables() contains the columns listed in Table 179 in the order given. The rows are ordered by TABLE_TYPE, TABLE_CAT, TABLE_SCHEM, and TABLE_NAME.

Since calls to SQLTables() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns may be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Table 179. Columns Returned By SQLTables

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	The name of the catalog containing TABLE_SCHEM. This column contains a NULL value.

Table 179. Columns Returned By SQLTables (continued)

Column Name	Data Type	Description
TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
TABLE_NAME	VARCHAR(128)	The name of the table, or view, or alias, or synonym.
TABLE_TYPE	VARCHAR(128)	Identifies the type given by the name in the TABLE_NAME column. It can have the string values 'TABLE', 'VIEW', 'INOPERATIVE VIEW', 'SYSTEM TABLE', 'ALIAS', or 'SYNONYM'.
REMARKS	VARCHAR(254)	Contains the descriptive information about the table.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 180. SQLTables SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled.	Asynchronous processing was enabled for the *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on the *StatementHandle*. Then the function was called again on the *StatementHandle*. The function was called and, before it completed execution, `SQLCancel()` was called on the *StatementHandle* from a different thread in a multithread application.
HY009	Invalid argument value.	*TableName* is null.

Table 180. SQLTables SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS. The valid of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the SQLGetInfo() function.
HYC00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. Timeouts are only supported on non-multitasking systems such as Windows 3.1 and Macintosh System 7. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetConnectAttr().

Restrictions

None.

Example

Also, refer to "Querying Environment Information Example" on page 39.

```
/* From CLI sample browser.c */
/* ... */
SQLRETURN init_tables( SQLHANDLE hstmt ) {

    SQLRETURN rc ;

    SQLUSMALLINT rowstat[MAX_TABLES];
    SQLUINTEGER pcrow;

    /* SQL_ROWSET_SIZE sets the max num of result rows to fetch each time */
```

```
rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_ROW_ARRAY_SIZE,
                     ( SQLPOINTER ) MAX_TABLES,
                     0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Set Size of One row, Used for Row-Wise Binding Only */
rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_ROW_BIND_TYPE,
                     ( SQLPOINTER ) sizeof( table_info ),
                     0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_ROW_STATUS_PTR,
                     ( SQLPOINTER ) rowstat,
                     0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_ROWS_FETCHED_PTR,
                     ( SQLPOINTER ) &pcrow,
                     0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

printf("Enter Search Pattern for Table Schema Name:\n");
gets((char *)table->schem);
printf("Enter Search Pattern for Table Name:\n");
gets((char *)table->name);
rc = SQLTables(hstmt, NULL, 0, table->schem, SQL_NTS,
               table->name, SQL_NTS, NULL, 0);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) table[0].schem, 129,
                &table[0].schem_l);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) table[0].name, 129,
                &table[0].name_l);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) table[0].type, 129,
                &table[0].type_l);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLBindCol(hstmt, 5, SQL_C_CHAR, (SQLPOINTER) table[0].remarks, 255,
                &table[0].remarks_l);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Now fetch the result set */
```

References

- "SQLColumns - Get Column Information for a Table" on page 315
- "SQLTablePrivileges - Get Privileges Associated With A Table" on page 740

SQLTransact - Transaction Management

Deprecated

Note:

In ODBC version 3, SQLTransact() has been deprecated and replaced with SQLEndTran(); see "SQLEndTran - End Transactions of a Connection" on page 356 for more information.

Although this version of DB2 CLI continues to support SQLTransact(), we recommend that you begin using SQLEndTran() in your DB2 CLI programs so that they conform to the latest standards.

See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information on this and other deprecated functions.

Migrating to the New Function

The statement:

```
SQLTransact(henv, hdbc, SQL_COMMIT);
```

for example, would be rewritten using the new function as:

```
SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT);
```

Purpose

Specification:	**DB2 CLI** 1.1	**ODBC** 1.0	**ISO CLI**

SQLTransact() commits or rolls back the current transaction in the specified connection. SQLTransact() can also be used to request that a commit or rollback be issued for each of the connections associated with the environment.

All changes to the database performed on the connection since connect time or the previous call to SQLTransact() (whichever is the most recent) are committed or rolled back.

If a transaction is active on a connection, the application must call SQLTransact() before it can disconnect from the database.

Syntax

```
SQLRETURN   SQLTransact      (SQLHENV          EnvironmentHandle, /* henv */
                              SQLHDBC          ConnectionHandle,  /* hdbc */
                              SQLUSMALLINT     Type);             /* fType */
```

Function Arguments

Table 181. SQLTransact Arguments

Data Type	Argument	Use	Description
SQLHENV	*EnvironmentHandle*	input	Environment handle. If *ConnectionHandle* is a valid connection handle, *EnvironmentHandle* is ignored.
SQLHDBC	*ConnectionHandle*	input	Database connection handle. If *ConnectionHandle* is set to SQL_NULL_HDBC, then *EnvironmentHandle* must contain the environment handle that the connection is associated with.
SQLUSMALLINT	*Type*	input	The desired action for the transaction. The value for this argument must be one of: • SQL_COMMIT • SQL_ROLLBACK

Usage

In DB2 CLI, a transaction begins implicitly when an application that does not already have an active transaction, issues `SQLPrepare()`, `SQLExecDirect()`, `SQLExecDirect()`, `SQLGetTypeInfo()`, or one of the catalog functions. The transaction ends when the application calls `SQLTransact()`.

If the input connection handle is SQL_NULL_HDBC and the environment handle is valid, then a commit or rollback will be issued on each of the open connections in the environment. SQL_SUCCESS is returned only if success is reported on all the connections. If the commit or rollback fails for one or more of the connections, `SQLTransact()` will return SQL_ERROR. To determine which connection(s) failed the commit or rollback operation, the application needs to call `SQLError()` on each connection handle in the environment.

It is important to note that unless the connection option SQL_ATTR_CONNECTTYPE has been set to SQL_COORDINATED_TRANS (to indicate coordinated distributed transactions), there is no attempt to provide coordinated global transaction with one-phase or two-phase commit protocols.

Completing a transaction has the following effects:

- Prepared SQL statements (via `SQLPrepare()`) survive transactions; they can be executed again without first calling `SQLPrepare()`.
- Cursor positions are maintained after a commit unless one or more of the following is true:

  - the server is SQL/DS
  - the SQL_ATTR_CURSOR_HOLD statement attribute for this handle is set to SQL_CURSOR_HOLD_OFF.
  - The CURSORHOLD keyword in the DB2 CLI initialization file is set so that cursor with hold is not in effect and this has not been overridden by resetting the SQL_ATTR_CURSOR_HOLD statement attribute.
  - The CURSORHOLD keyword is present in a the connection string on the SQLDriverConnect() call that set up this connection, and it indicates cursor with hold is not in effect, and this has not been overridden by resetting the SQL_ATTR_CURSOR_HOLD statement attribute.

  If the cursor position is not maintained due to any one of the above circumstances, the cursor is closed and all pending results are discarded.

  If the cursor position is maintained after a commit, the application must issue a fetch to re-position the cursor (to the next row) before continuing with processing of the remaining result set.

  To determine whether cursor position will be maintained after a commit, call SQLGetInfo() with the SQL_CURSOR_COMMIT_BEHAVIOR information type.
- Cursors are closed after a rollback and all pending results are discarded.
- Statement handles are still valid after a call to SQLTransact(), and can be reused for subsequent SQL statements or de-allocated by calling SQLFreeStmt().
- Cursor names, bound parameters, and column bindings survive transactions.

If no transaction is currently active on the connection, calling SQLTransact() has no effect on the database server and returns SQL_SUCCESS.

SQLTransact() may fail while executing the COMMIT or ROLLBACK due to a loss of connection. In this case the application may be unable to determine whether the COMMIT or ROLLBACK has been processed, and a database administrator's help may be required. Refer to the DBMS product information for more information on transaction logs and other transaction management tasks.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 182. SQLTransact SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection is closed.	The *ConnectionHandle* was not in a connected state.
08007	Connection failure during transaction.	The connection associated with the *ConnectionHandle* failed during the execution of the function during the execution of the function and it cannot be determined whether the requested COMMIT or ROLLBACK occurred before the failure.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function.
HY012	Invalid transaction code.	The value specified for the argument *Type* was neither SQL_COMMIT not SQL_ROLLBACK.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “SQLEndTran - End Transactions of a Connection” on page 356.

References

- “SQLSetStmtAttr - Set Options Related to a Statement” on page 702
- “SQLGetInfo - Get General Information” on page 489

Appendix A. Programming Hints and Tips

This section provides some hints and tips to help improve DB2 CLI and ODBC application performance and portability.

Setting Common Connection Attributes

The following connection attributes may need to be set (or considered) by DB2 CLI applications.

SQL_ATTR_AUTOCOMMIT

Generally this attribute should be set to SQL_AUTOCOMMIT_OFF, since each commit request can generate extra network flow. Only leave SQL_AUTOCOMMIT on if specifically needed.

Note: The default is SQL_AUTOCOMMIT_ON.

SQL_ATTR_TXN_ISOLATION

This connection attribute determines the isolation level at which the connection or statement will operate. The isolation level determines the level of concurrency possible, and the level of locking required to execute the statement. Applications need to choose an isolation level that maximizes concurrency, yet ensures data consistency.

Refer to the *SQL Reference* for a complete discussion of isolation levels and their effect.

Setting Common Statement Attributes

The following statement attributes may need to be set by DB2 CLI applications.

SQL_ATTR_MAX_ROWS

Setting this attribute limits the number of rows returned to the application. This can be used to avoid an application from being overwhelmed with a very large result set generated inadvertently, especially for applications on clients with limited memory resources.

In DB2 UDB v6, setting SQL_ATTR_MAX_ROWS will add the “OPTIMIZE for n ROWS” and “Fetch n Rows ONLY” clauses onto the statement. For other

servers, the full result set is still generated at the server, DB2 CLI will only fetch up to SQL_ATTR_MAX_ROWS rows.

SQL_ATTR_CURSOR_HOLD

This statement attribute determines if the cursor for this statement will be defined with the equivalent of the CURSOR WITH HOLD clause.

Resources associated with statement handles can be better utilized by DB2 CLI if the statements that do not require CURSOR WITH HOLD are set to SQL_CURSOR_HOLD_OFF.

Note: Many ODBC applications expect a default behavior where the cursor position is maintained after a commit.

SQL_ATTR_TXN_ISOLATION

DB2 CLI allows the isolation level to be set at the statement level, (however, we recommend that the isolation level be set at the connection level). The isolation level determines the level of concurrency possible, and the level of locking required to execute the statement.

Resources associated with statement handles can be better utilized by DB2 CLI if statements are set to the required isolation level, rather than leaving all statements at the default isolation level. This should only be attempted with a thorough understanding of the locking and isolation levels of the connected DBMS. Refer to the *SQL Reference* for a complete discussion of isolation levels and their effect.

Applications should use the minimum isolation level possible to maximize concurrency.

Comparing Binding and SQLGetData

Generally it is more efficient to bind application variables or file references to result sets than using `SQLGetData()`. Use `SQLGetData()`, or preferable, the LOB functions, when the data value is large variable-length data that:

- Must be received in pieces, or
- May not need to be retrieved (dependent on another application action)

Increasing Transfer Efficiency

The efficiency of transferring of character data between bound application variables and DB2 CLI can be increased if the *pcbValue* and *rgbValue* arguments are contiguous in memory. (This allows DB2 CLI to fetch both values with one copy operation.)

For example:

```
struct {  SQLINTEGER  pcbValue;
          SQLCHAR     rgbValue[MAX_BUFFER];
       } column;
```

Limiting Use of Catalog Functions

In general, try to limit the number of times the catalog functions are called, and limit the number of rows returned.

The number of catalog function calls can be reduced by calling the function once, and storing the information at the application.

The number of rows returned can be limited by specifying a:

- Schema name or pattern for all catalog functions
- Table name or pattern for all catalog functions other than `SQLTables`
- Column name or pattern for catalog functions that return detailed column information.

Remember, although an application may be developed and tested against a data source with hundreds of tables, it may be ran against a database with thousands of tables. Plan ahead.

Close any open cursors (call `SQLFreeStmt()` with SQL_CLOSE) for statement handles used for catalog queries to release any locks against the catalog tables. Outstanding locks on the catalog tables can prevent CREATE, DROP or ALTER statements from executing.

Using Column Names of Function Generated Result Sets

The column names of the result sets generated by catalog and information functions may change as the ODBC and CLI standards evolve. The *position* of the columns however, will not change.

Any application dependency should be based on the column position (*icol* parameter) and not the name.

Loading DB2 CLI Specific Functions From ODBC Applications

Call SQLGetInfo() with the SQL_DRIVER_HSTMT option to obtain the DB2 CLI statement handle (HSTMT).

The DB2 CLI functions can then be called directly from the shared library or DLL, using an operating system call and passing the HTSMT argument. (The ODBC driver manager maintains its own set of statement handles which are mapped to *actual* driver handles on each call. When a DB2 CLI function is called directly, this mapping must be done explicitly by calling SQLGetInfo().

Making use of Dynamic SQL Statement Caching

Note: This section is only relevant when connecting to a server that does not have a global statement cache. The application should not make use of dynamic SQL statement caching when connection to a server that has a global dynamic statement cache, such as DB2 Universal Database. See "Making use of the Global Dynamic Statement Cache" for more details.

To make use of *dynamic caching*, (when the server caches a prepared version of a dynamic SQL statement), the application must use the same statement handle for the same SQL statement.

For example, if an application routinely uses a set of 10 SQL statements, 10 statement handles should be allocated and associated with each of those statements. Do not free the statement handle while the statement may still be executed. (The transaction can still be rolled back or committed without affecting any of the prepared statements). The application would continue to prepare and execute the statements in a normal manner, DB2 CLI will determine if the prepare is actually needed.

To reduce function call overhead, the statement can be prepared once, and executed repeatedly throughout the application.

Note: If the server does not support dynamic caching, DB2 CLI will internally issue prepares for the statement when necessary.

Making use of the Global Dynamic Statement Cache

DB2 Universal Database version 5 or later has a *global dynamic statement cache* stored on the server. This cache is used to store the most popular access plans for prepared SQL statements.

Before each statement is prepared, the server automatically searches this cache to see if an access plan has already been created for this exact SQL statement (by this application or any other application or client). If so, the server does not need to generate a new access plan, but will use the one in the cache instead. There is now no need for the application to cache connections at the client unless connecting to a server that does not have a global dynamic statement cache (such as DB2 Common Server v2). For information on caching connections at the client see "Caching Statement Handles on the Client" on page 770 in the Migration section.

Optimizing Insertion and Retrieval of Data

The methods described in "Using Arrays to Input Parameter Values" on page 83 and "Retrieving a Result Set into an Array" on page 90 use compound SQL to optimize the network flow.

Use these methods as much as possible.

Optimizing for Large Object Data

Use LOB data types and the supporting functions for long strings whenever possible. Unlike LONG VARCHAR, LONG VARBINARY, and LONG VARGRAPHIC types, LOB data values can use LOB locators and functions such as `SQLGetPostion()` and `SQLGetSubString()` to manipulate large data values at the server.

LOB values can also be fetched directly to a file, and LOB parameter values can be read directly from a file. This saves the overhead of the application transferring data via application buffers.

Case Sensitivity of Object Identifiers

All database object (tables, views, columns etc.) identifiers are stored in the catalog tables in upper case unless the identifier is delimited. If an identifier is created using a delimited name, the exact case of the name is stored in the catalog tables.

When an identifier is referenced within an SQL statement, it is treated as case *insensitive* unless it is delimited.

For example, if the following two tables are created,

```
CREATE TABLE MyTable   (id INTEGER)
CREATE TABLE "YourTable" (id INTEGER)
```

two tables will exist, MYTABLE and YourTable

Both of the following statements are equivalent:

```
SELECT * FROM MyTable (id INTEGER)
SELECT * FROM MYTABLE (id INTEGER)
```

The second statement below will fail with TABLE NOT FOUND since there is no table named YOURTABLE:

```
SELECT * FROM "YourTable" (id INTEGER)   // executes without error
SELECT * FROM YourTable (id INTEGER)     // error, table not found
```

All DB2 CLI catalog function arguments treat the names of objects as *case sensitive*, that is, as if each name was delimited.

Using SQLDriverConnect Instead of SQLConnect

Using SQLDriverConnect() allows the application to rely on the dialog box provided by DB2 CLI to prompt the user for the connection information.

If an application uses its own dialog boxes to query the connect information, the user should be able to specify additional connect options in the connection string. The string should also be stored and used as a default on subsequent connections.

Implementing an SQL Governor

Each time an SQL statement is prepared, the server *estimates* the cost of the statement. The application can then decide whether to continue with the execution of the statement.

This estimate can be obtained from the SQLCA (SQLERRD(4)), and used by the application directly or the SQL_DB2ESTIMATE connect option can be set to a threshold value. If the estimated cost of any statement exceeds the threshold, DB2 CLI displays a dialog box, with a warning and a prompt to continue or cancel the execution of the statement.

The suggested threshold value is 60000, although in general the application should allow the end user to set the threshold value.

Note: The estimate is only an estimate of the total resources used by the server to execute the statement, it does not indicate the time required to execute the statement.

An estimate of the number of rows in the result is also available from the SQLCA (SQLERRD(3)), and could also be used by the application to restrict large queries.

Note: The accuracy of the information returned in the SQLERRD(3) and SQLERRD(4) fields is dependent on many factors such as the use of parameter markers and expressions within the statement. The main factor which can be controlled is the accuracy of the database statistics. That is, when the statistics were last updated, (for example, for DB2 Universal Database, the last time the RUNSTATS command was run.)

Turning Off Statement Scanning

DB2 CLI by default, scans each SQL statement searching for vendor escape clause sequences.

If the application does not generate SQL statements that contain vendor escape clause sequences (“Using Vendor Escape Clauses” on page 144), then the SQL_NO_SCAN statement option should be set to SQL_NOSCAN_ON at the connection level so that DB2 CLI does not perform a scan for vendor escape clauses.

Holding Cursors Across Rollbacks

Applications that need to deal with complex transaction management issues, may benefit from establishing multiple concurrent connections to the same database. Since each connection in DB2 CLI has its own transaction scope, any actions performed on one connection will not affect the transactions of other connections.

For example, all open cursors within a transaction get closed if a problem causes the transaction to be rolled back. An application can use multiple connections to the same database to separate statements with open cursors; since the cursors are in separate transactions, a rollback on one statement does not affect the cursors of the other statements.

Using multiple connections may mean bringing some data across to the client on one connection, and then sending it back to the server on the other connection. For example:

Suppose in connection #1 you are accessing Large Object columns and have created LOB locators that map to portions of large object values.

If in connection #2, you wish to use (e.g. insert) the portion of the LOB values represented by the LOB locators, you would have to move the LOB

values in connection #1 first to the application, and then pass them to the tables that you are working with in connection #2. This is because connection #2 does not know anything about the LOB locators in connection #1.

If you only had one connection, then you could just use the LOB locators directly. However, you would lose the LOB locators as soon as you rolled back your transaction.

Preparing Compound SQL Sub-Statements

In order to maximize efficiency of the compound statement, sub-statements should be prepared before the BEGIN COMPOUND statement, and then executed within the compound statement.

This also simplifies error handling since prepare errors can be handled outside of the compound statement.

Casting User Defined Types (UDTs)

If a parameter marker is used in a predicate of a query statement, and the parameter is a user defined type, the statement must use a CAST function to cast either the parameter marker or the UDT.

For example, if the following type and table is defined:

```
CREATE DISTINCT TYPE CNUM AS INTEGER WITH COMPARISONS

CREATE TABLE CUSTOMER (
        Cust_Num     CNUM NOT NULL,
        First_Name   CHAR(30) NOT NULL,
        Last_Name    CHAR(30) NOT NULL,
        Phone_Num    CHAR(20) WITH DEFAULT,
        PRIMARY KEY  (Cust_Num) )
```

This statement would fail since the parameter marker cannot be of type CNUM and thus the comparison fails due to incompatible types: SELECT first_name, last_name, phone_num FROM customer where cust_num = ?

Casting the column to integer (its base SQL type), allows the comparison to work since a parameter can be provided for type integer:

```
SELECT  first_name, last_name, phone_num from customer
where cast( cust_num as integer ) = ?
```

Alternatively the parameter marker can be cast to INTEGER and the server can then apply the INTEGER to CNUM conversion:

```
SELECT  first_name, last_name, phone_num FROM customer
where cust_num = cast( ? as integer )
```

Refer to the `custrep.c` sample file for a full working example.

Refer to the *SQL Reference* for more information about:

- Parameter markers, refer to the PREPARE statement
- casting, refer to the CAST function.

Use Multiple Threads rather than Asynchronous Execution

The asyncronous SQL model should only be used on non-threaded operating systems such a Windows 3.1. If your application cannot make use of multi-threading then see "Asynchronous Execution of CLI" on page 138.

Asynchronous SQL should NOT be used on platforms that support multiple threads.

"Writing Multi-Threaded Applications" on page 46 describes why and how to make use of multiple threads with DB2 CLI. Some common uses include:

- A thread other than the one executing can be used to call `SQLCancel()` (to cancel a long running query for example).
- Most GUI based applications use threads in order to ensure that user interaction can be handled on a higher priority thread than other application tasks, such as accessing the database.
- Executing DB2 CLI functions on multiple threads can improve throughput.

Using Deferred Prepare to Reduce Network Flow

In DB2 CLI Version 5, deferred prepare is on by default. The PREPARE request is not sent to the server until the corresponding execute request is issued. This minimizes network flow and improves performance.

See "Deferred Prepare now on by Default" on page 773 for complete details.

Appendix B. Migrating Applications

This section covers what has changed since the previous version of DB2 CLI, any incompatibilities and how to deal with them.

Summary of Changes

Version 5 of DB2 Universal Database contains new features which can affect the way you create DB2 CLI applications. The *SQL Reference* contains a complete summary of changes.

For a summary of DB2 CLI functions, and which version they were added, refer to "DB2 CLI Function Summary" on page 211.

There are many changes to DB2 CLI for version 5. Some of the highlights are listed below:

- A number of DB2 CLI functions have been deprecated. See "DB2 CLI Functions Deprecated for Version 5" on page 768 for more information.
- Some functions have been renamed (`SQLColAttributes()` is now `SQLColAttribute()`, for example).
- The three functions used to allocate handles (`SQLAllocConnect()`, `SQLAllocEnv()`, and `SQLAllocStmt()`) have been reduced to one multipurpose function (`SQLAllocHandle()`).
- A number of new DB2 CLI functions have been added to support the new features now available in DB2 CLI and ODBC (descriptors, scrollable cursors, etc...).

DB2 CLI also continues to contain extensions to access DB2 features that can not be accessed by ODBC applications. For example:

- Support for random access of Large Objects (LOBs), LOB locators and file reference buffers (sequential access is also possible).
- SQLCA access for detailed DB2 specific diagnostic information
- Control over null termination of output strings.

Incompatibilities

All Version 1 and Version 2 applications are binary compatible, meaning they will run with a DB2 Universal Database Version 5 product without change. In some cases, the DB2 CLI initialization file (db2cli.ini) may require customization in order for the applications to run. For more information about this file and the various keywords, refer to “Configuring db2cli.ini” on page 159.

If applications are recompiled, some minor changes may be required, these changes can be minimized by setting the DB2CLI_VER define, refer to “Setting the DB2CLI_VER Define” on page 778 for more information. The environment attribute SQL_ATTR_ODBC_VERSION can also be used to minimize the changes required. See the attribute description in “SQLSetEnvAttr - Set Environment Attribute” on page 682 for more information.

Deprecated Functions Not Supported in a 64-bit Environment

The following deprecated functions are not supported in a 64-bit environment:

- `SQLSetConnectOption()`
- `SQLSetStmtOption()`

They are not supported in a 64-bit environment because they have a 32-bit integer parameter that may be cast to a pointer (which is not possible on a 64-bit system).

Your application will have to be modified to use `SQLSetConnectAttr()` and `SQLSetStmtAttr()` before it can be used in a 64-bit environment.

Changes from Version 2.1.1 to 5.0.0

DB2 CLI Functions Deprecated for Version 5

Each DB2 CLI function that existed in version 2 but was deprecated in version 5 is still listed in the function reference, with an indication of its state at the beginning of the function description.

Although DB2 CLI version 5 continues to support all of the deprecated functions, we recommend that you begin using the new functions in your DB2 CLI programs so that they conform to the latest standards.

In some cases the features and arguments of the deprecated function and the replacement function are very similar (SQLColAttributes() and SQLColAttribute()). In these cases the description of the deprecated function has been removed.

In other cases the description of the deprecated function has been left to assist in understanding the conversion to the new function.

The following table lists each of the deprecated function, the type of description given, and the replacement function(s).

Table 183. Deprecated Functions and their Replacements

Deprecated Function	Description	Replacement Function
SQLAllocConnect()	Removed	SQLAllocHandle()
SQLAllocEnv()	Removed	SQLAllocHandle()
SQLAllocStmt()	Removed	SQLAllocHandle()
SQLColAttributes()	Removed	SQLColAttribute()
SQLError()	Unchanged	SQLGetDiagField() and SQLGetDiagRec()
SQLExtendedFetch()	Unchanged	SQLFetchScroll()
SQLFreeConnect()	Unchanged	SQLFreeHandle()
SQLFreeEnv()	Unchanged	SQLFreeHandle()
SQLGetConnectOption()	Removed	SQLGetConnectAttr()
SQLGetStmtOption()	Removed	SQLGetStmtAttr()
SQLParamOptions()	Unchanged	SQLSetStmtAttr()
SQLSetConnectOption()	Removed	SQLSetConnectAttr()
SQLSetParam()	Removed	SQLBindParameter()
SQLSetStmtOption()	Removed	SQLSetStmtAttr()
SQLTransact()	Unchanged	SQLEndTran()

Replacement of the Pseudo Catalog Table for Stored Procedures

DB2 Universal Database version 5 introduced two system catalog views used to store information about all stored procedures on the server. Before version 5, DB2 CLI used the pseudo catalog table for stored procedure registration. By default, DB2 CLI will use the new system catalog views. If the application expects to use the pseudo catalog table then the CLI/ODBC configuration keyword PATCH1 should be set to 262144.

In order for SQLProcedureColumns() and SQLProcedures() to return information about stored procedures (from the pseudo catalog table) when the application is connected to a version 2 DB2 common server, the pseudo

catalog table for stored procedure registration must have already been created and populated. This is the table named PROCEDURES in the DB2CLI schema. For further information on this pseudo catalog table, refer to and "Appendix H. Pseudo Catalog Table for Stored Procedure Registration" on page 839. It is imperative that the exact rules in "Appendix H. Pseudo Catalog Table for Stored Procedure Registration" on page 839 are followed when populating this table, or the SQLProcedureColumns() and SQLProcedures() calls will result in an error (SQLSTATE **42**601).

Setting a Subset of Statement Attributes using SQLSetConnectAttr()

The SQLSetConnectAttr() function can only be used to set a subset of statement attributes. This is being phased out, and new DB2 CLI should not rely on this feature. It was offered using SQLSetConnectOption() in versions of DB2 CLI prior to version 5, but it was then removed and the function was deprecated.

To support DB2 CLI applications written before version 5, SQLSetConnectAttr() currently behaves as SQLSetConnectOption() did using the option values defined in version 2 (the '_ATTR_' has been added for version 5):

- SQL_ATTR_ASYNC_ENABLE
- SQL_ATTR_BIND_TYPE
- SQL_ATTR_CONCURRENCY
- SQL_ATTR_CURSOR_HOLD
- SQL_ATTR_CURSOR_TYPE
- SQL_ATTR_MAX_LENGTH
- SQL_ATTR_MAX_ROWS
- SQL_ATTR_NODESCRIBE
- SQL_ATTR_NOSCAN
- SQL_ATTR_PARAMOPT_ATOMIC
- SQL_ATTR_QUERY_TIMEOUT
- SQL_ATTR_RETRIEVE_DATA
- SQL_ATTR_ROWSET_SIZE
- SQL_ATTR_STMTTXN_ISOLATION
- SQL_ATTR_TXN_ISOLATION

Caching Statement Handles on the Client

Previous versions of DB2 did not have the global statement cache. They did, however, provide *dynamic statement caching* at the server. In DB2 CLI terms this means that for a given statement handle, once a statement has been

prepared, it does not need to be prepared again (even after commits or rollbacks), so long as the statement handle is not freed. Applications that repeatedly execute the same SQL statement across multiple transactions, can save a significant amount of processing time and network traffic by:

1. Associating each such statement with its own statement handle, and
2. Preparing these statements once at the beginning of the application, then
3. Executing the statements as many times as is needed throughout the application.

This is not needed in DB2 Universal Database Version 5 and later because of the global dynamic statement cache. Preparing the first statement would create the package in the global cache. Each subsequent prepare request would find the first access plan in the cache and use it right away.

If an application is connected to a server that does not support dynamic statement caching across transaction boundaries, DB2 CLI will prepare each statement internally as needed. This means the method described above can be used for all applications, regardless of the RDBMS.

Changes to SQLColumns() Return Values

The following table lists the changes to the columns returned by `SQLColumns()` from version 2.1.1 to version 5.

For the current values see “Columns Returned by SQLColumns()” on page 317.

To have DB2 CLI behave as it did for version 2 (same column names and order), set the DB2 CLI/ODBC configuration keyword PATCH2. See “How to Set CLI/ODBC Configuration Keywords” on page 158 for more information on how to set this keyword.

Table 184. Changes to Columns Returned By SQLColumns

Version 2 Column	Version 5 Column	Change
14 DATETIME_CODE	Removed for version 5	This information is no longer returned. It was always NULL for version 2.
No version 2 equivalent	14 SQL_DATA_TYPE	This information was not returned in version 2.
No version 2 equivalent	15 SQL_DATETIME_SUB	This information was not returned in version 2.

Table 184. Changes to Columns Returned By SQLColumns (continued)

Version 2 Column	Version 5 Column	Change
15 CHAR_OCTET_LENGTH	16 CHAR_OCTET_LENGTH	The column number has changed, but the name and description remain the same.
16 ORDINAL_POSITION	17 ORDINAL_POSITION	The column number has changed, but the name and description remain the same.
17 IS_NULLABLE	18 IS_NULLABLE	The column number has changed, but the name and description remain the same.

Changes to SQLProcedureColumns() Return Values

The following table lists the changes to the columns returned by `SQLProcedureColumns()` from version 2.1.1 to version 5.

For the current values see "Columns Returned By SQLProcedureColumns" on page 596.

To have DB2 CLI behave at it did for version 2 (same column names and order), set the DB2 CLI/ODBC configuration keyword PATCH2. See "How to Set CLI/ODBC Configuration Keywords" on page 158 for more information on how to set this keyword.

Table 185. Changes to Columns Returned By SQLProcedureColumns

Version 2 Column	Version 5 Column	Change
14 ORDINAL_POSITION	18 ORDINAL_POSITION	The column number has changed, but the name and description remain the same.

Changes to the InfoTypes in SQLGetInfo()

Besides adding a number of new *InfoTypes* to `SQLGetInfo()`, the following version 2 values have been renamed for version 5:

Table 186. Changes to values used by SQLGetInfo()

Version 2 InfoType	Version 5 InfoType
SQL_ACTIVE_CONNECTIONS	SQL_MAX_DRIVER_CONNECTIONS
SQL_ACTIVE_STATEMENTS	SQL_MAX_CONCURRENT_ACTIVITIES

Table 186. Changes to values used by SQLGetInfo() (continued)

Version 2 InfoType	Version 5 InfoType
SQL_MAX_OWNER_NAME_LEN	SQL_MAX_SCHEMA_NAME_LEN
SQL_MAX_QUALIFIER_NAME_LEN	SQL_MAX_CATALOG_NAME_LEN
SQL_ODBC_SQL_OPT_IEF	SQL_INTEGRITY
SQL_SCHEMA_TERM	SQL_OWNER_TERM
SQL_OWNER_USAGE	SQL_SCHEMA_USAGE
SQL_QUALIFIER_LOCATION	SQL_CATALOG_LOCATION
SQL_QUALIFIER_NAME_SEPARATOR	SQL_CATALOG_NAME_SEPARATOR
SQL_QUALIFIER_TERM	SQL_CATALOG_TERM
SQL_QUALIFIER_USAGE	SQL_CATALOG_USAGE

Deferred Prepare now on by Default

In DB2 CLI Version 5, deferred prepare is on by default. The PREPARE request is not sent to the server until the corresponding execute request is issued. The two requests are then combined into one command/reply flow (instead of two) to minimize network flow and to improve performance. This is of greatest benefit when the application generates queries where the answer set is very small, and the overhead of separate requests and replies is not spread across multiple blocks of query data. In an environment where a DB2 Connect or DDCS gateway is used, there is a greater opportunity for cost reduction because four request and reply combinations are reduced to two.

DB2 CLI Version 2 applications that expect the PREPARE to be executed as soon as it requested may not operate as expected. (An application may, for instance, rely on the row and cost estimates that are normally returned in the SQLERRD(3) and SQLERRD(4) of the SQLCA of a prepare statement; with deferred prepare, these values may become zeros.) To ensure these programs work as they did with Version 2, the DB2 CLI/ODBC configuration keyword DEFERREDPREPARE can be set to disable deferred prepare. See “DEFERREDPREPARE” on page 182 for more information.

The statement attribute SQL_ATTR_DEFERRED_PREPARE can also be used to force DB2 CLI to prepare the statement as soon as it is issued. See the attribute in “SQLSetStmtAttr - Set Options Related to a Statement” on page 702 for more information.

In version 2, a DB2 CLI application could use the function `SQLSetColAttributes()` to reduce network traffic by describing the result descriptor information for every column in the result set. With deferred prepare this is no longer of any benefit, and the `SQLSetColAttributes()`

function has been deprecated. If your application does call this function it will ignore all arguments, and will always return SQL_SUCCESS.

Changes from version 2.1.0 to 2.1.1

Stored Procedures that return multi-row result sets

Previous versions of DB2 CLI did not support multi-row result sets. Version 2.1.1 provides the ability to retrieve one or more result sets from a stored procedure call by leaving one or more cursors open, each associated with a query, when the stored procedure exists.

Data Conversion and Values for SQLGetInfo

DB2 CLI version 2.1.1 now supports a convert function defined by ODBC using vendor escape clauses. This function will convert between chars (CHAR, VARCHAR, LONG VARCHAR and CLOB), and DOUBLE (or FLOAT).

DB2 CLI version 2.1.0 returned zero for all SQL_CONVERT *fInfoTypes* using `SQLGetInfo()`. Now that version 2.1.1 supports conversion, `SQLGetInfo()` returns a set of bitmasks for the *fInfoTypes* that start with SQL_CONVERT_ (SQL_CONVERT_INTEGER for example) which can be used for comparison with the bitmasks that start with SQL_CVT_ (SQL_CVT_CHAR for example).

In addition to the CONVERT function, DB2 CLI version 2.1.1 provides two new date and time functions that can be accessed using the ODBC vendor escape clause convention:

JULIAN_DAY(day_expr)
Returns an integer corresponding to the number of days in date_exp relative to January 1, 4712 B.C.

SECONDS_SINCE_MIDNIGHT(time_expr)
Returns an integer corresponding to the number of seconds in time_expr relative to midnight.

Changes from version 1.x to 2.1.0

The following describes the difference between version 2.1.0 and 1.x.

AUTOCOMMIT and CURSOR WITH HOLD Defaults

Previous versions of DB2 CLI did not support autocommit (each statement is a transaction), and was thus equivalent to having AUTOCOMMIT set to **off**. DB2 CLI Version 2.1 now supports autocommit, but in order to be consistent with ODBC, the default autocommit behavior is **on**.

To enable existing DB2 CLI applications to run with the same behavior as previous versions, set the AUTOCOMMIT keyword to `0`. This keyword applies to all applications; for this reason new applications should explicitly override the keyword and set the autocommit connection option to the required value.

Support for specifying CURSOR WITH HOLD was also added in Version 2.1, with a default of *with hold* **on**. Since this was not supported in previous versions, with hold was effectively set **off**.

Although this change does not affect the behavior of applications as much as autocommit, the CURSORHOLD keyword should be set to `0` (cursors are not maintained after a commit). This keyword applies to all applications, for this reason new applications should explicitly override the keyword and set the CURSORHOLD statement option to the required value.

Graphic Data Type Values

The `#define` values

- SQL_GRAPHIC
- SQL_VARGRAPHIC
- SQL_LONGVARGRAPHIC

have been changed in Version 2.1 in order that they can be used with ODBC applications. DB2 CLI will still accept the old values, but it is recommended that existing applications that use these valued be recompiled.

These valued are defined in the `sqlcli.h` header file.

SQLSTATES

In previous versions, DB2 CLI returned the **S1**009 SQLSTATE instead of the more explicit **S1**090 to **S1**110 series of SQLSTATES defined by ODBC.

As a result of X/Open also using this range of SQLSTATES, DB2 CLI Version 2.1 will also return the more explicit SQLSTATES.

Mixing Embedded SQL, Without CONNECT RESET

DB2 CLI's Version 2.1 support of multiple connections may affect existing applications that mix the use of embedded SQL and DB2 CLI.

If your application:

1. Connects to a database using embedded SQL (including using the command line processor or Administrative APIs).
2. (Does NOT issue a reset).

3. Connect to a database using DB2 CLI

the second connect will fail since it is not same type of connection as the first connect.

The application must issue a CONNECT RESET before calling a DB2 CLI connect function.

Note: An application should always explicitly reset a connection.

Use of VARCHAR FOR BIT DATA

Character data defined with the FOR BIT DATA clause is associated with a default C buffer type of SQL_C_BINARY. If data is defined as FOR BIT DATA, it is transferred to:

- SQL_C_BINARY buffers unchanged
- SQL_C_CHAR buffers as a character representation of the hexadecimal value of the data. Each byte is represented by two ASCII characters, (this means the SQL_C_CHAR buffer must be double the size of the FOR BIT DATA string.)

Existing applications that explicitly use SQL_C_CHAR with data defined as FOR BIT DATA, will get a different result and may receive only half of the original data. The initialization keyword, BITDATA, can be set to 0 to force DB2 CLI to treat FOR BIT DATA in the same was as previous versions.

User Defined Types in Predicates

Existing applications may be affected if tables are modified to make use of User Defined Types.

If a parameter marker is used in a predicate of a query statement, and the parameter is a user defined type, the statement must use a CAST function to cast either the parameter marker or the UDT.

For example, if the following type and table is defined:

```
CREATE DISTINCT TYPE CNUM AS INTEGER WITH COMPARISONS

CREATE TABLE CUSTOMER (
        Cust_Num     CNUM NOT NULL,
        First_Name   CHAR(30) NOT NULL,
        Last_Name    CHAR(30) NOT NULL,
        Phone_Num    CHAR(20) WITH DEFAULT,
        PRIMARY KEY  (Cust_Num) )
```

The following statement would fail since the parameter marker cannot be of type CNUM and thus the comparison fails due to incompatible types:

```
SELECT  first_name, last_name, phone_num FROM customer
where cust_num = ?
```

Casting the column to integer (its base SQL type), allows the comparison to work since a parameter can be provided for type integer:

```
SELECT  first_name, last_name, phone_num from customer
where cast( cust_num as integer ) = ?
```

Alternatively the parameter marker can be cast to INTEGER. This informs the server of the parameter marker's type and allows the default INTEGER to CNUM conversion to be applied:

```
SELECT  first_name, last_name, phone_num FROM customer
where cust_num = cast( ? as integer )
```

Refer to the `custrep.c` sample file for a full working example.

Refer to the *SQL Reference* for more information about:

- UDT, refer to CREATE DISTINCT TYPE
- Parameter markers, refer to the PREPARE statement
- casting, refer to the CAST function.

Data Conversion Values for SQLGetInfo

In versions prior to Version 2.1, DB2 CLI returned a set of bitmasks for the *fInfoTypes* which started with SQL_CONVERT_ (for example, SQL_CONVERT_INTEGER). *fInfoTypes* which where used with corresponding comparison bitmasks which started with SQL_CVT_ (for example, SQL_CVT_CHAR).

Since this *fInfoType* is defined by ODBC to indicate supported conversion functions, and these functions are not supported, DB2 CLI Version 2.1 now (correctly) returns zero for all SQL_CONVERT *fInfoTypes*.

Function Prototype Changes

In order to better align with X/Open and ODBC, some DB2 CLI function arguments have changed to unsigned data types, and two new types have been introduced, SQLUINTEGER and SQLUSMALLINT.

DB2 CLI applications prior to version 2.1 will generate complier errors for mismatched function arguments when compiled with version 2.1 header files unless DB2CLI_VER is defined, see "Setting the DB2CLI_VER Define" on page 778.

We recommend that existing source files eventually be modified to declare the necessary arguments using the SQLUINTEGER and SQLUSMALLINT types.

Setting the DB2CLI_VER Define

The DB2CLI_VER define allows the application to specify that the DB2 CLI header files are to remain compatible with previous versions of DB2 CLI.

DB2CLI_VER must be set to a hex value either as a compile flag, or as a `#define` before the DB2 CLI header files are included. For example, using the `-D` compiler flag:

```
-DDB2CLI_VER=0x0110
```

sets the DB2CLI_VER to Version 1.1.

In Version 2.1, if DB2CLI_VER is not defined it defaults to 0x0210.

Appendix C. DB2 CLI and ODBC

This appendix discusses the support provided by the DB2 ODBC driver, and how it differs from DB2 CLI.

Figure 18 on page 780 below compares DB2 CLI and the DB2 ODBC driver.

1. an ODBC driver under the ODBC Driver Manager
2. DB2 CLI, callable interface designed for DB2 specific applications.

DB2 Client refers to all available DB2 Clients. DB2 refers to all DB2 Universal Database products.

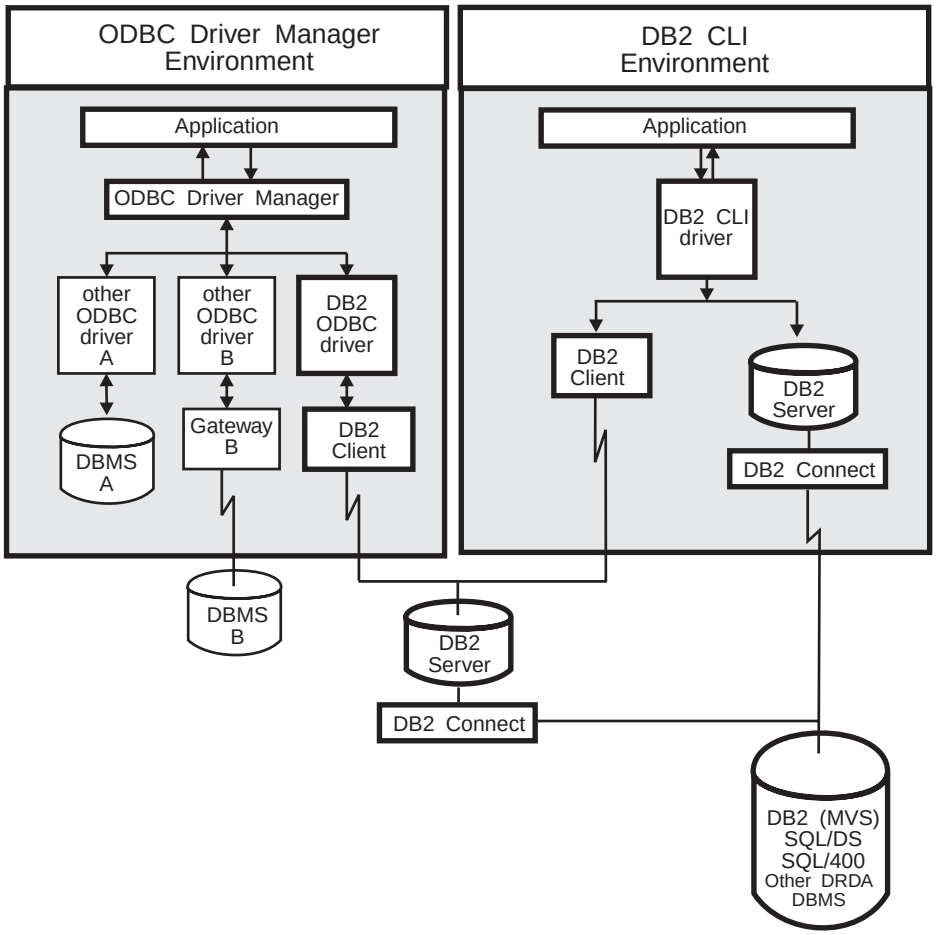


Figure 18. DB2 CLI and ODBC.

In an ODBC environment, the Driver Manager provides the interface to the application. It also dynamically loads the necessary *driver* for the database server that the application connects to. It is the driver that implements the ODBC function set, with the exception of some extended functions implemented by the Driver Manager. In this environment DB2 CLI conforms to level 2 of ODBC 2.0, and level 1 of ODBC 3.0. In addition it also conforms to the following ODBC 3.0 level 2 interface conformance items:

202 Dynamic parameters described using SQLDescribeParam

203 Support for input, output, and input/output parameters as well as result values of stored procedures

205 Advanced information on the data dictionary from SQLColumnPriviledges, SQLForeignKeys, and SQLTablePrivileges

207 Asynchronous execution of appropriate functions

209 SQL_ATTR_CONCURRENCY statement attribute can be set to at least one value other than SQL_CONCUR_READ_ONLY

211 Default isolation level can be changed. Transactions can be executed with the "serialized" level of isolation.

For ODBC application development, you must obtain an ODBC Software Development Kit (from Microsoft for Microsoft platforms, and from other vendors for non-Microsoft platforms.) When developing ODBC applications that may connect to DB2 servers, use this book (for information on DB2 specific extensions and diagnostic information) in conjunction with the *ODBC 3.0 Software Development Kit and Programmer's Reference.*

In environments without an ODBC driver manager, DB2 CLI is a self sufficient driver which supports a subset of the functions provided by the ODBC driver. Table 187 summarizes the two levels of support, and Table 12 on page 211 provides a complete list of ODBC 3.0 functions, and indicates if they are supported.

Table 187. DB2 CLI ODBC Support

ODBC Features	**DB2 ODBC Driver**	**DB2 CLI**
Core Level Functions	All	All
Level 1 Functions	All	All
Level 2 Functions	All	All, except for `SQLDrivers()`
Additional DB2 CLI Functions	All, functions can be accessed by dynamically loading the DB2 CLI library, see "Appendix A. Programming Hints and Tips" on page 757 for more information.	• `SQLSetConnection()` • `SQLGetEnvAttr()` • `SQLSetEnvAttr()` • `SQLSetColAttributes()` • `SQLGetSQLCA()` • `SQLBindFileToCol()` • `SQLBindFileToParam()` • `SQLExtendedBind()` • `SQLExtendedPrepare()` • `SQLGetLength()` • `SQLGetPosition()` • `SQLGetSubString()`

Table 187. DB2 CLI ODBC Support (continued)

ODBC Features	DB2 ODBC Driver	DB2 CLI
SQL Data Types	All the types listed for DB2 CLI, as well as: • SQL_BINARY • SQL_VARBINARY • SQL_LONGVARBINARY	• SQL_BIGINT • SQL_BINARY • SQL_BLOB • SQL_BLOB_LOCATOR • SQL_CHAR • SQL_CLOB • SQL_CLOB_LOCATOR • SQL_DBCLOB • SQL_DBCLOB_LOCATOR • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC

Table 187. DB2 CLI ODBC Support (continued)

ODBC Features	DB2 ODBC Driver	DB2 CLI
C Data Types	All the types listed for DB2 CLI.	• SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DATE • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TIME • SQL_C_TIMESTAMP • SQL_C_TINYINT • SQL_C_SBIGINT • SQL_C_UBIGINT • SQL_C_NUMERIC ** **** Only supported on 32-bit Windows**
Return Codes	All the codes listed for DB2 CLI.	• SQL_SUCCESS • SQL_SUCCESS_WITH_INFO • SQL_STILL_EXECUTING • SQL_NEED_DATA • SQL_NO_DATA_FOUND • SQL_ERROR • SQL_INVALID_HANDLE
SQLSTATES	Mapped to X/Open SQLSTATES with additional IBM SQLSTATES, with the exception of the ODBC type **08**S01.	Mapped to X/Open SQLSTATES with additional IBM SQLSTATES
Multiple connections per application	Supported	Supported
Dynamic loading of driver	Supported	Not applicable

For more information on ODBC refer to *ODBC 3.0 Software Development Kit and Programmer's Reference.*

ODBC Function List

Table 12 on page 211 is a complete list of all Microsoft's ODBC 3.0 functions. The ODBC conformance level and whether it is supported by DB2 CLI is shown for each function.

For a complete list of DB2 CLI functions, and information about X/Open and ISO callable SQL standards, refer to "DB2 CLI Function Summary" on page 211.

Isolation Levels

The following table map IBM RDBMs isolation levels to ODBC transaction isolation levels. The `SQLGetInfo()` function, indicates which isolation levels are available.

Table 188. Isolation Levels Under ODBC

IBM Isolation Level	**ODBC Isolation Level**
Cursor Stability	SQL_TXN_READ_COMMITTED
Repeatable Read	SQL_TXN_SERIALIZABLE_READ
Read Stability;	SQL_TXN_REPEATABLE_READ
Uncommitted Read	SQL_TXN_READ_UNCOMMITTED
No Commit	(no equivalent in ODBC)
Note: `SQLSetConnectOption()` and `SQLSetStmtOption` will return SQL_ERROR with an SQLSTATE of **HY**009 if you try to set an unsupported isolation level.	

Appendix D. Extended Scalar Functions

The following functions are defined by ODBC using vendor escape clauses. Each function may be called using the escape clause syntax, or calling the equivalent DB2 function.

These functions are presented in the following categories:

For more information about vendor escape clauses, refer to "ODBC Scalar Functions" on page 147.

The tables in the following sections indicates for which servers (and the earliest versions) that the function can be accessed, when called from an application using DB2 CLI Version 2.1.

All errors detected by the following functions, when connected to a DB2 Version 2 server, will return SQLSTATE 38552. The text portion of the message is of the form SYSFUN:*nn* where *nn* is one of the following reason codes:

01 Numeric value out of range

02 Division by zero

03 Arithmetic overflow or underflow

04 Invalid date format

05 Invalid time format

06 Invalid timestamp format

07 Invalid character representation of a timestamp duration

08 Invalid interval type (must be one of 1, 2, 4, 8, 16, 32, 64, 128, 256)

09 String too long

10 Length or position in string function out of range

11 Invalid character representation of a floating point number

String Functions

The string functions in this section are supported by DB2 CLI and defined by ODBC using vendor escape clauses.

- Character string literals used as arguments to scalar functions must be bounded by single quotes.
- Arguments denoted as *string_exp* can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR, or SQL_CLOB.
- Arguments denoted as *start, length, code* or *count* can be a numeric literal or the result of another scalar function, where the underlying data type is integer based (SQL_SMALLINT, SQL_INTEGER).
- The first character in the string is considered to be at position 1.

Table 189. String Scalar Functions

ASCII(*string_exp* **)**
Returns the ASCII code value of the leftmost character of *string_exp* as an integer.

DB2 for	common server 2.1			

CHAR(*code* **)**
Returns the character that has the ASCII code value specified by *code*. The value of *code* should be between 0 and 255; otherwise, the return value is null.

DB2 for	common server 2.1			

CONCAT(*string_exp1***,** *string_exp2* **)**
Returns a character string that is the result of concatenating *string_exp2* to *string_exp1*.

DB2 for	common server 1.1	MVS	VM/VSE	AS/400

DIFFERENCE(*string_exp1, string_exp2* **)**
Returns an integer value indicating the difference between the values returned by the SOUNDEX function for *string_exp1* and *string_exp2*.

DB2 for	common server 1.1			

INSERT(*string_exp1, start, length, string_exp2* **)**
Returns a character string where *length* number of characters beginning at *start* has been replaced by *string_exp2* which contains *length* characters.

DB2 for	common server 1.1	MVS	VM/VSE	AS/400

Table 189. String Scalar Functions (continued)

LCASE(*string_exp* **)**
Converts all upper case characters in *string_exp* to lower case.

DB2 for	common server 1.0		VM/VSE	

LEFT(*string_exp,count* **)**
Returns the leftmost *count* of characters of *string_exp*.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

LENGTH(*string_exp* **)**
Returns the number of characters in *string_exp*, excluding trailing blanks and the string termination character.
Note: Trailing blanks were included prior to Version 2.1. Trailing blanks are still included for DB2 for MVS/ESA.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

LOCATE(*string_exp1***,** *string_exp2* **[,***start* **])**
Returns the starting position of the first occurrence of *string_exp1* within *string_exp2*. The search for the first occurrence of *string_exp1* begins with first character position in *string_exp2* unless the optional argument, *start*, is specified. If *start* is specified, the search begins with the character position indicated by the value of *start*. The first character position in *string_exp2* is indicated by the value 1. If *string_exp1* is not found withing *string_exp2*, the value 0 is returned.

DB2 for	common server 2.1			

LTRIM(*string_exp* **)**
Returns the characters of *string_exp* with the leading blanks removed.

DB2 for	common server 2.1	VM/VSE		AS/400

REPEAT(*string_exp***,** *count* **)**
Returns a character string composed of *string_exp* repeated *count* times.

DB2 for	common server 2.1	MVS	VM/VSE	AS/400

REPLACE(*string_exp1***,** *string_exp2***,** *string_exp3* **)**
Replaces all occurrences of *string_exp2* in *string_exp1* with *string_exp3*.

DB2 for	common server 2.1			

Table 189. String Scalar Functions (continued)

RIGHT(*string_exp*, *count* **)**
Returns the rightmost count of characters of *string_exp*.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

RTRIM(*string_exp* **)**
Returns the characters of *string_exp* with trailing blanks removed.

DB2 for	common server 2.1		VM/VSE	AS/400

SOUNDEX(*string_exp1* **)**
Returns a four character code representing the sound of *string_exp1*. Note that different data sources use different algorithms to represent the sound of *string_exp1*.

DB2 for	common server 1.1			

SPACE(*count* **)**
Returns a character string consisting of *count* spaces.

DB2 for	common server 2.1			

SUBSTRING(*string_exp*, *start*, *length* **)**
Returns a character string that is derived from *string_exp* beginning at the character position specified by *start* for *length* characters.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

UCASE(*string_exp* **)**
Converts all lower case characters in *string_exp* to upper case.

DB2 for	common server 1.0		VM/VSE	AS/400

Numeric Functions

The numeric functions in this section are supported by DB2 CLI and defined by ODBC using vendor escape clauses.

- Arguments denoted as *numeric_exp* can be the name of a column, the result of another scalar function, or a numeric literal, where the underlying data type can be either floating point based (SQL_NUMERIC, SQL_DECIMAL, SQL_FLOAT, SQL_REAL, SQL_DOUBLE) or integer based (SQL_SMALLINT, SQL_INTEGER).

- Arguments denoted as *double_exp* can be the name of a column, the result of another scalar functions, or a numeric literal where the underlying data type is floating point based.
- Arguments denoted as *integer_exp* can be the name of a column, the result of another scalar functions, or a numeric literal, where the underlying data type is integer based.

Table 190. Numeric Scalar Functions

ABS(*numeric_exp* **)**
Returns the absolute value of *numeric_exp*.

DB2 for	common server 2.1			AS/400

ACOS(*double_exp* **)**
Returns the arccosine of *double_exp* as an angle, expressed in radians.

DB2 for	common server 2.1			AS/400

ASIN(*double_exp* **)**
Returns the arcsine of *double_exp* as an angle, expressed in radians.

DB2 for	common server 2.1			AS/400

ATAN(*double_exp* **)**
Returns the arctangent of *double_exp* as an angle, expressed in radians.

DB2 for	common server 2.1			AS/400

ATAN2(*double_exp1***,** *double_exp2* **)**
Returns the arctangent of *x* and *y* coordinates specified by *double_exp1* and *double_exp2*, respectively, as an angle expressed in radians.

DB2 for	common server 2.1			

CEILING(*numeric_exp* **)**
Returns the smallest integer greater than or equal to *numeric_exp*.

DB2 for	common server 2.1			

COS(*double_exp* **)**
Returns the cosine of *double_exp*, where *double_exp* is an angle expressed in radians.

DB2 for	common server 2.1			AS/400

Table 190. Numeric Scalar Functions (continued)

COT(*double_exp* **)**
Returns the cotangent of *double_exp*, where *double_exp* is an angle expressed in radians.

DB2 for	common server 2.1			AS/400

DEGREES(*numeric_exp* **)**
Returns the number of degrees converted from *numeric_exp* radians.

DB2 for	common server 2.1			AS/400 3.6

EXP(*double_exp* **)**
Returns the exponential value of *double_exp*.

DB2 for	common server 2.1			AS/400

FLOOR(*numeric_exp* **)**
Returns the largest integer less than or equal to *numeric_exp*.

DB2 for	common server 2.1			AS/400 3.6

LOG(*double_exp* **)**
Returns the natural logarithm of *double_exp*.

DB2 for	common server 2.1			AS/400

LOG10(*double_exp* **)**
Returns the base 10 logarithm of *double_exp*.

DB2 for	common server 2.1			AS/400

MOD(*integer_exp1***,** *integer_exp2* **)**
Returns the remainder (modulus) of *integer_exp1* divided by *integer_exp2*.

DB2 for	common server 2.1			AS/400

PI()
Returns the constant value of pi as a floating point value.

DB2 for	common server 2.1			AS/400

POWER(*numeric_exp***,** *integer_exp* **)**
Returns the value of *numeric_exp* to the power of *integer_exp*.

Table 190. Numeric Scalar Functions (continued)

DB2 for	common server 2.1			AS/400 3.6

RADIANS(*numeric_exp* **)**
Returns the number of radians converted from *numeric_exp* degrees.

DB2 for	common server 2.1			

RAND([*integer_exp* **])**
Returns a random floating point value using *integer_exp* as the optional seed value.

DB2 for	common server 2.1			

ROUND(*numeric_exp***,** *integer_exp.* **)**
Returns *numeric_exp* rounded to *integer_exp* places right of the decimal point. If *integer_exp* is negative, *numeric_exp* is rounded to | *integer_exp* | places to the left of the decimal point.

DB2 for	common server 2.1			

SIGN(*numeric_exp* **)**
Returns an indicator or the sign of *numeric_exp*. If *numeric_exp* is less than zero, -1 is returned. If *numeric_exp* equals zero, 0 is returned. If *numeric_exp* is greater than zero, 1 is returned.

DB2 for	common server 2.1			

SIN(*double_exp* **)**
Returns the sine of *double_exp*, where *double_exp* is an angle expressed in radians.

DB2 for	common server 2.1			AS/400

SQRT(*double_exp* **)**
Returns the square root of *double_exp*.

DB2 for	common server 2.1			AS/400

TAN(*double_exp* **)**
Returns the tangent of *double_exp*, where *double_exp* is an angle expressed in radians.

DB2 for	common server 2.1			AS/400

Table 190. Numeric Scalar Functions (continued)

TRUNCATE(*numeric_exp*, *integer_exp* **)**
Returns *numeric_exp* truncated to *integer_exp* places right of the decimal point. If *integer_exp* is negative, *numeric_exp* is truncated to | *integer_exp* | places to the left of the decimal point.

DB2 for	common server 2.1			

Date and Time Functions

The date and time functions in this section are supported by DB2 CLI and defined by ODBC using vendor escape clauses.

- Arguments denoted as *timestamp_exp* can be the name of a column, the result of another scalar function, or a time,date, or timestamp literal.
- Arguments denoted as *date_exp* can be the name of a column, the result of another scalar function, or a date or timestamp literal, where the underlying data type can be character based, or date or timestamp based.
- Arguments denoted as *time_exp* can be the name of a column, the result of another scalar function, or a time or timestamp literal, where the underlying data types can be character based, or time or timestamp based.

Table 191. Date and Time Scalar Functions

CURDATE()
Returns the current date as a date value.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

CURTIME()
Returns the current local time as a time value.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

DAYNAME(*date_exp* **)**
Returns a character string containing the name of the day (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday) for the day portion of *date_exp*.

DB2 for	common server 2.1			

DAYOFMONTH (*date_exp* **)**
Returns the day of the month in *date_exp* as an integer value in the range of 1-31.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

Table 191. Date and Time Scalar Functions (continued)

DAYOFWEEK(*date_exp* **)**
Returns the day of the week in *date_exp* as an integer value in the range 1-7, where 1 represents Sunday.

DB2 for	common server 2.1			AS/400 3.6

DAYOFYEAR(*date_exp* **)**
Returns the day of the year in *date_exp* as an integer value in the range 1-366.

DB2 for	common server 2.1			AS/400 3.6

HOUR(*time_exp* **)**
Returns the hour in *time_exp* as an integer value in the range of 0-23.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

JULIAN_DAY(*date_exp* **)**
Returns the number of days between *date_exp* and January 1, 4712 B.C. (the start of the Julian date calendar). (When moving from a v2.1.0 to a v2.1.1 database you must run the migrate utility to access this function.)

DB2 for	common server 2.1.1			

MINUTE(*time_exp* **)**
Returns the minute in *time_exp* as integer value in the range of 0-59.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

MONTH(*date_exp* **)**
Returns the month in *date_exp* as an integer value in the range of 1-12.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

MONTHNAME(*date_exp* **)**
Returns a character string containing the name of month (January, February, March, April, May, June, July, August, September, October, November, December) for the month portion of *date_exp*.

DB2 for	common server 2.1			

NOW()
Returns the current date and time as a timestamp value.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

Table 191. Date and Time Scalar Functions (continued)

QUARTER(*date_exp* **)**
Returns the quarter in *date_exp* as an integer value in the range of 1-4.

DB2 for	common server 2.1			AS/400 3.6

SECOND(*time_exp* **)**
Returns the second in *time_exp* as an integer value in the range of 0-59.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

SECONDS_SINCE_MIDNIGHT(*time_exp* **)**
Returns the number of seconds in *time_exp* relative to midnight as an integer value in the range of 0-86400. If *time_exp* includes a fractional seconds component, the fractional seconds component will be discarded. (When moving from a v2.1.0 to a v2.1.1 database you must run the migrate utility to access this function.)

DB2 for	common server 2.1.1			

TIMESTAMPADD(*interval*, *integer_exp*, *timestamp_exp* **)**
Returns the timestamp calculated by adding *integer_exp* intervals of type *interval* to *timestamp_exp*. Valid values of interval are:

- SQL_TSI_FRAC_SECOND
- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR

where fractional seconds are expressed in billionths of a second. If *timestamp_exp* specifies a time value and *interval* specifies days, weeks, months, quarters, or years, the date portion of *timestamp_exp* is set to the current date before calculating the resulting timestamp. If *timestamp_exp* is a date value and *interval* specifies fractional seconds, seconds, minutes, or hours, the time portion of *timestamp_exp* is set to 00:00:00.000000 before calculating the resulting timestamp. An application determines which intervals are supported by calling ***SQLGetInfo()*** with the SQL_TIMEDATE_ADD_INTERVALS option.

DB2 for	common server 2.1			

Table 191. Date and Time Scalar Functions (continued)

TIMESTAMPDIFF(*interval*, *timestamp_exp1*, *timestamp_exp2* **)**

Returns the integer number of intervals of type *interval* by which *timestamp_exp2* is greater than *timestamp_exp1*. Valid values of interval are:

- SQL_TSI_FRAC_SECOND
- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR

where fractional seconds are expressed in billionths of a second. If either timestamp expression is a time value and *interval* specifies days, weeks, months, quarters, or years, the date portion of that timestamp is set to the current date before calculating the difference between the timestamps. If either timestamp expression is a date value and *interval* specifies fractional seconds, seconds, minutes, or hours, the time portion of that timestamp is set to 0 before calculating the difference between the timestamps. An application determines which intervals are supported by calling ***SQLGetInfo()*** with the SQL_TIMEDATE_DIFF_INTERVALS option.

DB2 for	common server 2.1			

WEEK(*date_exp* **)**

Returns the week of the year in *date_exp* as an integer value in the range of 1-54.

DB2 for	common server 2.1			AS/400 3.6

YEAR(*date_exp* **)**

Returns the year in *date_exp* as an integer value in the range of 1-9999.

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

For those functions that return a character string containing the name of the day of week or the name of the month, these character strings will be National Language Support enabled.

System Functions

The system functions in this section are supported by DB2 CLI and defined by ODBC using vendor escape clauses.

- Arguments denoted as *exp* can be the name of a column, the result of another scalar function, or a literal.
- Arguments denoted as *value* can be a literal constant.

Table 192. System Scalar Functions

DATABASE()
Returns the name of the database corresponding to the connection handle (*hdbc*). (The name of the database is also available via ***SQLGetInfo()*** by specifying the information type SQL_DATABASE_NAME.)

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

IFNULL(*exp*, *value* **)**
If *exp* is null, *value* is returned. If *exp* is not null, *exp* is returned. The possible data type(s) of *value* must be compatible with the data type of *exp*.

DB2 for	common server 1.2	MVS	VM/VSE	AS/400

USER()
Returns the user's authorization name. (The user's authorization name is also available via ***SQLGetInfo()*** by specifying the information type SQL_USER_NAME.)

DB2 for	common server 1.0	MVS	VM/VSE	AS/400

Conversion Function

The conversion function is supported by DB2 CLI and defined by ODBC using vendor escape clauses.

Each driver and datasource determines which conversions are valid between the possible data types. As the driver translates the ODBC syntax into native syntax it will reject the conversions that are not supported by the data source, even if the ODBC syntax is valid.

Use the function ***SQLGetInfo()*** with the appropriate convert function masks to determine which conversions are supported by the data source.

Table 193. Conversion Function

CONVERT(*expr_value, data_type* **)**

- *data_type* indicates the data type of the converted representation of *expr_value*, and can be either SQL_CHAR or SQL_DOUBLE.
- *expr_value* is the value to convert. It can be of various types, depending on the conversions supported by the driver and datasource. Use the function ***SQLGetInfo()*** with the appropriate convert function masks to determine which conversions are supported by the data source.

(When moving from a v2.1.0 to a v2.1.1 database you must run the migrate utility to access this function.)

DB2 for	common server 2.1			

Appendix E. SQLSTATE Cross Reference

This table is a cross-reference of all the SQLSTATEs listed in the *Diagnostics* section of each function description in Chapter 5. DB2 CLI Functions.

Note: DB2 CLI may also return SQLSTATEs generated by the server that are not listed in this table. If the returned SQLSTATE is not listed here, refer to the documentation for the server for additional SQLSTATE information.

SQLState Cross Reference table

01000 (Warning.)

- SQLAllocHandle()
- SQLBrowseConnect()
- SQLBuildDataLink()
- SQLBulkOperations()
- SQLCloseCursor()
- SQLColAttribute()
- SQLCopyDesc()
- SQLDescribeParam()
- SQLEndTran()
- SQLExtendedPrepare()
- SQLFetchScroll()
- SQLFreeHandle()
- SQLGetConnectAttr()
- SQLGetDataLinkAttr()
- SQLGetDescField()
- SQLGetDescRec()
- SQLGetStmtAttr()
- SQLSetConnectAttr()
- SQLSetDescField()
- SQLSetDescRec()
- SQLSetPos()
- SQLSetStmtAttr()

01002 (Disconnect error.)

- SQLDisconnect()

01004 (Data truncated.)

- SQLBrowseConnect()
- SQLBuildDataLink()
- SQLBulkOperations()
- SQLColAttribute()
- SQLDataSources()
- SQLDescribeCol()
- SQLDriverConnect()
- SQLExtendedFetch()
- SQLFetch()
- SQLFetchScroll()
- SQLGetConnectAttr()
- SQLGetCursorName()
- SQLGetData()
- SQLGetDataLinkAttr()
- SQLGetDescField()
- SQLGetDescRec()
- SQLGetInfo()
- SQLGetStmtAttr()
- SQLGetSubString()
- SQLNativeSql()
- SQLPutData()
- SQLSetPos()

01504 (The UPDATE or DELETE statement does not include a WHERE clause.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

01508 (Statement disqualified for blocking.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

01S00 (Invalid connection string attribute.)

- SQLBrowseConnect()
- SQLDriverConnect()

01S01 (Error in row.)

- SQLExtendedFetch()
- SQLFetchScroll()
- SQLSetPos()
- SQLBulkOperations()

01S02 (Option value changed.)

- SQLBrowseConnect()
- SQLSetConnectAttr()
- SQLSetDescField()
- SQLSetStmtAttr()
- SQLExtendedPrepare()

01S06 (Attempt to fetch before the result set returned the first rowset.)

- SQLFetchScroll()

01S07 (Fractional truncation.)

- SQLFetchScroll()
- SQLSetPos()
- SQLBulkOperations()

07001 (Wrong number of parameters.)

- SQLExecDirect()

07002 (Too many columns.)

- SQLExtendedFetch()
- SQLFetch()
- SQLFetchScroll()

07005 (The statement did not return a result set.)

- SQLColAttribute()
- SQLDescribeCol()

07006 (Invalid conversion.)

- SQLBindParameter()
- SQLBulkOperations()
- SQLExecDirect()
- SQLExtendedBind()
- SQLExtendedFetch()
- SQLFetch()
- SQLFetchScroll()
- SQLGetData()
- SQLGetLength()
- SQLGetPosition()
- SQLGetSubString()
- SQLParamData()
- SQLSetPos()

07009 (Invalid descriptor index)

- SQLBindCol()
- SQLBulkOperations()
- SQLColAttribute()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLExtendedBind()
- SQLFetch()
- SQLFetchScroll()
- SQLGetDescField()
- SQLGetDescRec()
- SQLSetDescField()
- SQLSetDescRec()
- SQLSetPos()

08001 (Unable to connect to data source.)

- SQLBrowseConnect()
- SQLConnect()

08002 (Connection in use.)

- SQLBrowseConnect()
- SQLConnect()
- SQLSetConnectAttr()

08003 (Connection is closed.)

- SQLAllocHandle()
- SQLDisconnect()
- SQLEndTran()
- SQLGetConnectAttr()
- SQLGetInfo()
- SQLNativeSql()
- SQLSetConnectAttr()
- SQLSetConnection()
- SQLTransact()

08004 (The application server rejected establishment of the connection.)

- SQLBrowseConnect()
- SQLConnect()

08007 (Connection failure during transaction.)

- SQLEndTran()
- SQLTransact()

08S01 (Communication link failure.)

- SQLBrowseConnect()
- SQLCopyDesc()
- SQLDescribeParam()
- SQLExtendedPrepare()
- SQLFetchScroll()
- SQLFreeHandle()
- SQLGetDescField()
- SQLGetDescRec()
- SQLSetConnectAttr()
- SQLSetDescField()
- SQLSetDescRec()
- SQLSetStmtAttr()

0F001 (The LOB token variable does not currently represent any value.)

- SQLGetLength()
- SQLGetPosition()
- SQLGetSubString()

21S01 (Insert value list does not match column list.)

- SQLDescribeParam()
- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

21S02 (Degrees of derived table does not match column list.)

- SQLExecDirect()
- SQLPrepare()
- SQLSetPos()
- SQLBulkOperations()
- SQLExtendedPrepare()

22001 (String data right truncation.)

- SQLExecDirect()
- SQLFetchScroll()
- SQLPutData()
- SQLSetPos()
- SQLBulkOperations()

22002 (Invalid output or indicator buffer specified.)

- SQLExtendedFetch()
- SQLFetch()
- SQLFetchScroll()
- SQLGetData()

22003 (Numeric value out of range.)

- SQLBulkOperations()
- SQLExecDirect()
- SQLExtendedFetch()
- SQLFetch()
- SQLFetchScroll()
- SQLGetData()
- SQLPutData()
- SQLSetPos()

22005 (Error in assignment.)

- SQLExecDirect()
- SQLExtendedFetch()
- SQLFetch()
- SQLGetData()
- SQLPutData()

22007 (Invalid datetime format.)

- SQLBulkOperations()
- SQLExecDirect()
- SQLExtendedFetch()
- SQLFetch()
- SQLFetchScroll()
- SQLGetData()
- SQLPutData()
- SQLSetPos()

22008 (Datetime field overflow.)

- SQLBulkOperations()
- SQLExecDirect()
- SQLExtendedFetch()
- SQLFetch()
- SQLGetData()
- SQLSetPos()

22011 (A substring error occurred.)

- SQLGetSubString()

22012 (Division by zero is invalid.)

- SQLExecDirect()
- SQLExtendedFetch()
- SQLFetch()
- SQLFetchScroll()

22015 (Interval field overflow)

- SQLBulkOperations()

22018 (Invalid character value for cast specification.)

- SQLFetchScroll()
- SQLPrepare()
- SQLBulkOperations()
- SQLExtendedPrepare()

22019 (Invalid escape character)

- SQLPrepare()
- SQLExtendedPrepare()

22025 (Invalid escape sequence)

- SQLPrepare()
- SQLExtendedPrepare()

22026 (String data, length mismatch)

- SQLParamData()

23000 (Integrity constraint violation.)

- SQLExecDirect()
- SQLBulkOperations()

24000 (Invalid cursor state.)

- SQLBulkOperations()
- SQLCloseCursor()
- SQLColumnPrivileges()
- SQLColumns()
- SQLExecDirect()
- SQLExtendedFetch()
- SQLExtendedPrepare()
- SQLFetch()
- SQLFetchScroll()
- SQLForeignKeys()
- SQLGetData()
- SQLGetStmtAttr()
- SQLGetTypeInfo()
- SQLPrepare()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLSetConnectAttr()
- SQLSetStmtAttr()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()

24504 (The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.)

- SQLExecDirect()

2500025501 (Invalid transaction state.)

- SQLDisconnect()

25501 (Invalid transaction state.)

- SQLDisconnect()

28000 (Invalid authorization specification.)

- SQLBrowseConnect()
- SQLConnect()

34000 (Invalid cursor name.)

- SQLExecDirect()
- SQLPrepare()
- SQLSetCursorName()
- SQLExtendedPrepare()

37000 (Invalid SQL syntax.)

- SQLNativeSql()

37xxx (Invalid SQL syntax.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

40001 (Serialization failure)

- SQLBulkOperations()
- SQLColumnPrivileges()
- SQLEndTran()
- SQLExecDirect()
- SQLExtendedPrepare()
- SQLFetchScroll()
- SQLParamData()
- SQLPrepare()

40003 (Statement completion unknown)

- SQLBulkOperations()

4000308S01 (Communication link failure.)

- SQLBindCol()
- SQLBindFileToCol()
- SQLBindFileToParam()
- SQLBindParameter()
- SQLCancel()
- SQLColumnPrivileges()
- SQLColumns()
- SQLDescribeCol()
- SQLExecDirect()
- SQLExtendedBind()
- SQLExtendedFetch()
- SQLExtendedPrepare()
- SQLFetch()
- SQLForeignKeys()
- SQLFreeStmt()
- SQLGetCursorName()
- SQLGetData()
- SQLGetFunctions()
- SQLGetInfo()
- SQLGetLength()
- SQLGetPosition()
- SQLGetSubString()
- SQLGetTypeInfo()
- SQLMoreResults()
- SQLNumParams()
- SQLNumResultCols()
- SQLParamData()
- SQLParamOptions()
- SQLPrepare()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLPutData()
- SQLRowCount()
- SQLSetCursorName()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()

42000 (Syntax error or access violation)

- SQLBulkOperations()

42601 (PARMLIST syntax error.)

- SQLProcedureColumns()

42818 (The operands of an operator or function are not compatible.)

- SQLGetPosition()

42895 (The value of a host variable in the EXECUTE or OPEN statement cannot be used because of its data type.)

- SQLExecDirect()

428A1 (Unable to access a file referenced by a host file variable.)

- SQLExecDirect()
- SQLExtendedFetch()
- SQLFetch()

42xxx (Syntax Error or Access Rule Violation.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

44000 (Integrity constraint violation.)

- SQLExecDirect()
- SQLBulkOperations()

54028 (The maximum number of concurrent LOB handles has been reached.)

- SQLExtendedFetch()
- SQLFetch()

56084 (LOB data is not supported in DRDA.)

- SQLExecDirect()
- SQLExtendedFetch()
- SQLFetch()

58004 (Unexpected system failure.)

- SQLBindCol()
- SQLBindFileToCol()
- SQLBindFileToParam()
- SQLBindParameter()
- SQLConnect()
- SQLDataSources()
- SQLDescribeCol()
- SQLDisconnect()
- SQLExecDirect()
- SQLExtendedBind()
- SQLExtendedFetch()
- SQLExtendedPrepare()
- SQLFetch()
- SQLFreeConnect()
- SQLFreeEnv()
- SQLFreeStmt()
- SQLGetCursorName()
- SQLGetData()
- SQLGetFunctions()
- SQLGetInfo()
- SQLGetLength()
- SQLGetPosition()
- SQLGetSubString()
- SQLMoreResults()
- SQLNumResultCols()
- SQLPrepare()
- SQLRowCount()
- SQLSetCursorName()
- SQLTransact()

HY000 (General error.)

- SQLAllocHandle()
- SQLBrowseConnect()
- SQLBuildDataLink()
- SQLBulkOperations()
- SQLCloseCursor()
- SQLColAttribute()
- SQLCopyDesc()
- SQLDataSources()
- SQLDescribeParam()
- SQLDriverConnect()
- SQLEndTran()
- SQLExtendedPrepare()
- SQLFetchScroll()
- SQLFreeHandle()
- SQLGetConnectAttr()
- SQLGetDataLinkAttr()
- SQLGetDescField()
- SQLGetDescRec()
- SQLGetStmtAttr()
- SQLParamData()
- SQLSetConnectAttr()
- SQLSetConnection()
- SQLSetDescField()
- SQLSetDescRec()
- SQLSetPos()
- SQLSetStmtAttr()

HY001 (Memory allocation failure.)

- SQLAllocHandle()
- SQLBindCol()
- SQLBindFileToCol()
- SQLBindFileToParam()
- SQLBindParameter()
- SQLBrowseConnect()
- SQLBuildDataLink()
- SQLBulkOperations()
- SQLCancel()
- SQLCloseCursor()
- SQLColAttribute()
- SQLColumnPrivileges()
- SQLColumns()
- SQLConnect()
- SQLCopyDesc()
- SQLDataSources()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLDisconnect()
- SQLEndTran()
- SQLExecDirect()
- SQLExtendedBind()
- SQLExtendedFetch()
- SQLExtendedPrepare()
- SQLFetch()
- SQLFetchScroll()
- SQLForeignKeys()
- SQLFreeConnect()
- SQLFreeEnv()
- SQLFreeHandle()
- SQLFreeStmt()
- SQLGetConnectAttr()
- SQLGetCursorName()
- SQLGetData()
- SQLGetDataLinkAttr()
- SQLGetDescField()
- SQLGetDescRec()
- SQLGetEnvAttr()
- SQLGetFunctions()
- SQLGetInfo()
- SQLGetLength()
- SQLGetPosition()
- SQLGetStmtAttr()
- SQLGetSubString()
- SQLGetTypeInfo()
- SQLMoreResults()
- SQLNativeSql()
- SQLNumParams()
- SQLNumResultCols()
- SQLParamData()
- SQLParamOptions()
- SQLPrepare()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLPutData()
- SQLRowCount()
- SQLSetConnectAttr()
- SQLSetCursorName()
- SQLSetDescField()
- SQLSetDescRec()
- SQLSetPos()
- SQLSetStmtAttr()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()
- SQLTransact()

HY002 (Invalid column number.)

- SQLBindCol()
- SQLBindFileToCol()
- SQLDescribeCol()
- SQLGetData()

HY003 (Program type out of range.)

- SQLBindCol()
- SQLBindParameter()
- SQLExtendedBind()
- SQLGetData()
- SQLGetLength()
- SQLGetSubString()

HY004 (SQL data type out of range.)

- SQLBindFileToParam()
- SQLGetTypeInfo()
- SQLBindParameter()
- SQLExtendedBind()

HY007 (Associated statement is not prepared.)

- SQLCopyDesc()
- SQLGetDescField()
- SQLGetDescRec()

HY008 (Operation was cancelled)

- SQLBulkOperations()
- SQLColAttribute()
- SQLColumnPrivileges()
- SQLColumns()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLExtendedPrepare()
- SQLFetch()
- SQLFetchScroll()
- SQLNumParams()
- SQLNumResultCols()
- SQLParamData()
- SQLPrepare()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLPutData()
- SQLSetPos()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()

HY009 (Invalid argument value.)

- SQLBindFileToCol()
- SQLBindFileToParam()
- SQLBindParameter()
- SQLColumnPrivileges()
- SQLExecDirect()
- SQLExtendedBind()
- SQLExtendedPrepare()
- SQLForeignKeys()
- SQLGetDataLinkAttr()
- SQLGetLength()
- SQLGetPosition()
- SQLGetSubString()
- SQLNativeSql()
- SQLPrepare()
- SQLPutData()
- SQLSetConnectAttr()
- SQLSetCursorName()
- SQLSetEnvAttr()
- SQLSetStmtAttr()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()

HY010 (Function sequence error.)

- SQLAllocHandle()
- SQLBindCol()
- SQLBindFileToCol()
- SQLBindFileToParam()
- SQLBindParameter()
- SQLBulkOperations()
- SQLCloseCursor()
- SQLColAttribute()
- SQLColumnPrivileges()
- SQLColumns()
- SQLCopyDesc()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLDisconnect()
- SQLEndTran()
- SQLExecute()
- SQLExtendedBind()
- SQLExtendedFetch()
- SQLExtendedPrepare()
- SQLFetch()
- SQLFetchScroll()
- SQLForeignKeys()
- SQLFreeConnect()
- SQLFreeEnv()
- SQLFreeHandle()
- SQLFreeStmt()
- SQLGetConnectAttr()
- SQLGetCursorName()
- SQLGetData()
- SQLGetDescField()
- SQLGetDescRec()
- SQLGetFunctions()
- SQLGetLength()
- SQLGetPosition()
- SQLGetStmtAttr()
- SQLGetSubString()
- SQLGetTypeInfo()
- SQLMoreResults()
- SQLNumParams()
- SQLNumResultCols()
- SQLParamData()
- SQLParamOptions()
- SQLPrepare()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLPutData()
- SQLRowCount()
- SQLSetConnectAttr()
- SQLSetCursorName()
- SQLSetDescField()
- SQLSetDescRec()
- SQLSetPos()
- SQLSetStmtAttr()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()

HY011 (Operation invalid at this time.)

- SQLBulkOperations()
- SQLExtendedPrepare()
- SQLSetConnectAttr()
- SQLSetEnvAttr()
- SQLSetPos()
- SQLSetStmtAttr()

HY012 (Invalid transaction code.)

- SQLEndTran()
- SQLTransact()

HY013 (Unexpected memory handling error.)

- SQLAllocHandle()
- SQLBindCol()
- SQLBindFileToCol()
- SQLBindFileToParam()
- SQLBindParameter()
- SQLBrowseConnect()
- SQLBulkOperations()
- SQLCancel()
- SQLCloseCursor()
- SQLConnect()
- SQLDataSources()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLDisconnect()
- SQLExecDirect()
- SQLExtendedBind()
- SQLExtendedFetch()
- SQLExtendedPrepare()
- SQLFetch()
- SQLFreeConnect()
- SQLFreeEnv()
- SQLFreeHandle()
- SQLGetCursorName()
- SQLGetData()
- SQLGetDescField()
- SQLGetDescField()
- SQLGetDescRec()
- SQLGetFunctions()
- SQLGetLength()
- SQLGetPosition()
- SQLGetStmtAttr()
- SQLGetSubString()
- SQLMoreResults()
- SQLNumParams()
- SQLNumResultCols()
- SQLParamData()
- SQLPrepare()
- SQLRowCount()
- SQLSetCursorName()
- SQLSetDescRec()
- SQLTransact()

HY014 (No more handles.)

- SQLAllocHandle()
- SQLColumnPrivileges()
- SQLColumns()
- SQLExecDirect()
- SQLExtendedPrepare()
- SQLForeignKeys()
- SQLPrepare()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()

HY016 (Cannot modify an implementation row descriptor.)

- SQLCopyDesc()
- SQLSetDescField()
- SQLSetDescRec()

HY017 (Invalid use of an automatically allocated descriptor handle.)

- SQLFreeHandle()
- SQLSetStmtAttr()
- SQLExtendedPrepare()

HY018 (Server declined cancel request.)

- SQLCancel()

HY021 (Inconsistent descriptor information.)

- SQLBindParameter()
- SQLCopyDesc()
- SQLExtendedBind()
- SQLGetDescField()
- SQLSetDescField()
- SQLSetDescRec()

HY024 (Invalid attribute value.)

- SQLSetConnectAttr()
- SQLSetEnvAttr()
- SQLSetStmtAttr()
- SQLExtendedPrepare()

HY090 (Invalid string or buffer length.)

- SQLBindCol()
- SQLBindFileToCol()
- SQLBindFileToParam()
- SQLBindParameter()
- SQLBrowseConnect()
- SQLBuildDataLink()
- SQLBulkOperations()
- SQLColAttribute()
- SQLColumnPrivileges()
- SQLColumns()
- SQLConnect()
- SQLDataSources()
- SQLDescribeCol()
- SQLDriverConnect()
- SQLExecDirect()
- SQLExtendedBind()
- SQLExtendedPrepare()
- SQLForeignKeys()
- SQLGetConnectAttr()
- SQLGetCursorName()
- SQLGetData()
- SQLGetDataLinkAttr()
- SQLGetDescField()
- SQLGetInfo()
- SQLGetPosition()
- SQLGetStmtAttr()
- SQLGetSubString()
- SQLNativeSql()
- SQLPrepare()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLPutData()
- SQLSetConnectAttr()
- SQLSetCursorName()
- SQLSetEnvAttr()
- SQLSetPos()
- SQLSetStmtAttr()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()

HY091 (Descriptor type out of range.)

- SQLColAttribute()
- SQLGetDescField()
- SQLSetDescField()

HY092 (Option type out of range.)

- SQLAllocHandle()
- SQLBulkOperations()
- SQLCopyDesc()
- SQLEndTran()
- SQLExecDirect()
- SQLExtendedFetch()
- SQLExtendedPrepare()
- SQLFetch()
- SQLFreeStmt()
- SQLGetConnectAttr()
- SQLGetDataLinkAttr()
- SQLGetEnvAttr()
- SQLGetStmtAttr()
- SQLParamData()
- SQLSetConnectAttr()
- SQLSetDescField()
- SQLSetEnvAttr()
- SQLSetPos()
- SQLSetStmtAttr()

HY093 (Invalid parameter number.)

- SQLBindFileToParam()
- SQLBindParameter()
- SQLExtendedBind()

HY094 (Invalid scale value.)

- SQLBindParameter()
- SQLExtendedBind()

HY096 (Information type out of range.)

- SQLGetInfo()

HY097 (Column type out of range.)

- SQLSpecialColumns()

HY098 (Scope type out of range.)

- SQLSpecialColumns()

HY099 (Nullable type out of range.)

- SQLSpecialColumns()

HY100 (Uniqueness option type out of range.)

- SQLStatistics()

HY101 (Accuracy option type out of range.)

- SQLStatistics()

HY103 (Direction option out of range.)

- SQLDataSources()

HY104 (Invalid precision value.)

- SQLBindParameter()
- SQLExtendedBind()

HY105 (Invalid parameter type.)

- SQLSetDescField()
- SQLBindParameter()
- SQLExtendedBind()

HY106 (Fetch type out of range.)

- SQLExtendedFetch()
- SQLFetchScroll()

HY107 (Row value out of range.)

- SQLFetchScroll()
- SQLParamOptions()
- SQLSetPos()

HY109 (Invalid cursor position.)

- SQLGetStmtAttr()
- SQLSetPos()

HY110 (Invalid driver completion.)

- SQLDriverConnect()

HY111 (Invalid bookmark value.)

- SQLFetchScroll()

HY501 (Invalid data source name.)

- SQLConnect()

HY503 (Invalid file name length.)

- SQLExecDirect()

HY506 (Error closing a file.)

- SQLCancel()
- SQLFreeStmt()
- SQLParamData()

HY509 (Error deleting a file.)

- SQLParamData()

HYC00 (Driver not capable.)

- SQLAllocHandle()
- SQLBindCol()
- SQLBindFileToCol()
- SQLBindFileToParam()
- SQLBindParameter()
- SQLBulkOperations()
- SQLColAttribute()
- SQLColumnPrivileges()
- SQLColumns()
- SQLDescribeCol()
- SQLExtendedBind()
- SQLExtendedFetch()
- SQLExtendedPrepare()
- SQLFetch()
- SQLFetchScroll()
- SQLForeignKeys()
- SQLGetConnectAttr()
- SQLGetData()
- SQLGetInfo()
- SQLGetLength()
- SQLGetPosition()
- SQLGetStmtAttr()
- SQLGetSubString()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLSetConnectAttr()
- SQLSetEnvAttr()
- SQLSetPos()
- SQLSetStmtAttr()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()

HYT00 (Timeout expired.)

- SQLBulkOperations()
- SQLColumnPrivileges()
- SQLColumns()
- SQLConnect()
- SQLDescribeCol()
- SQLExecDirect()
- SQLExtendedFetch()
- SQLExtendedPrepare()
- SQLFetch()
- SQLForeignKeys()
- SQLGetData()
- SQLGetTypeInfo()
- SQLMoreResults()
- SQLNumParams()
- SQLNumResultCols()
- SQLParamData()
- SQLPrepare()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLPutData()
- SQLSetPos()
- SQLSpecialColumns()
- SQLStatistics()
- SQLTablePrivileges()
- SQLTables()

HYT01 (Connection timeout expired)

- SQLBulkOperations()

S0001 (Database object already exists.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

S0002 (Database object does not exist.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

S0011 (Index already exists.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

S0012 (Index not found.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

S0021 (Column already exists.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

S0022 (Column not found.)

- SQLExecDirect()
- SQLPrepare()
- SQLExtendedPrepare()

Appendix F. Data Conversion

This section contains tables used for data conversion between C and SQL data types. This includes:

- Precision, scale, length, and display size of each data type
- Conversion from SQL to C data types
- Conversion from C to SQL data types

For a list of SQL and C data types, their symbolic types, and the default conversions, refer to Table 2 on page 29 AND Table 3 on page 31. Supported conversions are shown in Table 6 on page 35.

Data Type Attributes

Information is shown for the following Data Type Attributes:

- "Precision"
- "Scale" on page 818
- "Length" on page 819
- "Display Size" on page 820

Precision

The precision of a numeric column or parameter refers to the maximum number of digits used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter. The following table defines the precision for each SQL data type.

Table 194. Precision

fSqlType	Precision
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length of the column or parameter. For example, the precision of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column or parameter. [a]
SQL_DECIMAL SQL_NUMERIC	The defined maximum number of digits. For example, the precision of a column defined as NUMERIC(10,3) is 10.
SQL_SMALLINT [b]	5

Table 194. Precision (continued)

fSqlType	Precision
SQL_BIGINT	19
SQL_INTEGER [b]	10
SQL_FLOAT [b]	15
SQL_REAL [b]	7
SQL_DOUBLE [b]	15
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length of the column or parameter. For example, the precision of a column defined as CHAR(10) FOR BIT DATA, is 10.
SQL_LONGVARBINARY	The maximum length of the column or parameter.
SQL_DATE [b]	10 (the number of characters in the yyyy-mm-dd format).
SQL_TIME [b]	8 (the number of characters in the hh:mm:ss format).
SQL_TIMESTAMP	The number of characters in the "yyy-mm-dd hh:mm:ss[.fff[fff]]" format used by the TIMESTAMP data type. For example, if a timestamp does not use seconds or fractional seconds, the precision is 16 (the number of characters in the "yyyy-mm-dd hh:mm" format). If a timestamp uses thousandths of a second, the precision is 23 (the number of characters in the "yyyy-mm-dd hh:mm:ss.fff" format).
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column or parameter. For example, the precision of a column defined as GRAPHIC(10) is 10.
SQL_LONGVARGRAPHIC	The maximum length of the column or parameter.

Note:

a When defining the precision of a parameter of this data type with `SQLBindParameter()` or `SQLSetParam()`, *cbParamDef* should be set to the total length of the data, not the precision as defined in this table.

b The *cbParamDef* argument of `SQLBindParameter()` or `SQLSetParam()` is ignored for this data type.

Scale

The scale of a numeric column or parameter refers to the maximum number of digits to the right of the decimal point. Note that, for approximate floating point number columns or parameters, the scale is undefined, since the number of digits to the right of the decimal place is not fixed. The following table defines the scale for each SQL data type.

Table 195. Scale

fSqlType	Scale
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Not applicable.
SQL_DECIMAL SQL_NUMERIC	The defined number of digits to the right of the decimal place. For example, the scale of a column defined as NUMERIC(10, 3) is 3.
SQL_SMALLINT SQL_INTEGER SQL_BIGINT	0
SQL_REAL SQL_FLOAT SQL_DOUBLE	Not applicable.
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Not applicable.
SQL_DATE SQL_TIME	Not applicable.
SQL_TIMESTAMP	The number of digits to the right of the decimal point in the "yyyy-mm-dd hh:mm:ss[fff[fff]]" format. For example, if the TIMESTAMP data type uses the "yyyy-mm-dd hh:mm:ss.fff" format, the scale is 3.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC SQL_DBCLOB	Not applicable.

Length

The length of a column is the maximum number of *bytes* returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte. Note that the length of a column may be different than the number of bytes required to store the data on the data source. For a list of default C data types, see the "Default C Data Types" section.

The following table defines the length for each SQL data type.

Table 196. Length

fSqlType	Length
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length of the column. For example, the length of a column defined as CHAR(10) is 10.

Table 196. Length (continued)

fSqlType	Length
SQL_LONGVARCHAR	The maximum length of the column.
SQL_DECIMAL SQL_NUMERIC	The maximum number of digits plus two. Since these data types are returned as character strings, characters are needed for the digits, a sign, and a decimal point. For example, the length of a column defined as NUMERIC(10,3) is 12.
SQL_SMALLINT	2 (two bytes).
SQL_INTEGER	4 (four bytes).
SQL_REAL	4 (four bytes).
SQL_FLOAT	8 (eight bytes).
SQL_DOUBLE	8 (eight bytes).
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length of the column. For example, the length of a column defined as CHAR(10) FOR BIT DATA is 10.
SQL_LONGVARBINARY	The maximum length of the column.
SQL_DATE SQL_TIME	6 (the size of the DATE_STRUCT or TIME_STRUCT structure).
SQL_TIMESTAMP	16 (the size of the TIMESTAMP_STRUCT structure).
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column times 2. For example, the length of a column defined as GRAPHIC(10) is 20.
SQL_LONGVARGRAPHIC	The maximum length of the column times 2.

Display Size

The display size of a column is the maximum number of *bytes* needed to display data in character form. The following table defines the display size for each SQL data type.

Table 197. Display Size

fSqlType	Display Size
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length of the column. For example, the display size of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column.
SQL_DECIMAL SQL_NUMERIC	The precision of the column plus two (a sign, precision digits, and a decimal point). For example, the display size of a column defined as NUMERIC(10,3) is 12.
SQL_SMALLINT	6 (a sign and 5 digits).

Table 197. Display Size (continued)

fSqlType	Display Size
SQL_INTEGER	11 (a sign and 10 digits).
SQL_BIGINT	20 (a sign and 19 digits).
SQL_REAL	13 (a sign, 7 digits, a decimal point, the letter E, a sign, and 2 digits).
SQL_FLOAT SQL_DOUBLE	22 (a sign, 15 digits, a decimal point, the letter E, a sign, and 3 digits).
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length of the column times 2 (each binary byte is represented by a 2 digit hexadecimal number). For example, the display size of a column defined as CHAR(10) FOR BIT DATA is 20.
SQL_LONGVARBINARY	The maximum length of the column times 2.
SQL_DATE	10 (a date in the format yyyy-mm-dd).
SQL_TIME	8 (a time in the format hh:mm:ss).
SQL_TIMESTAMP	19 (if the scale of the timestamp is 0) or 20 plus the scale of the timestamp (if the scale is greater than 0). This is the number of characters in the "yyyy-mm-dd hh:mm:ss[fff[fff]]" format. For example, the display size of a column storing thousandths of a second is 23 (the number of characters in "yyyy-mm-dd hh:mm:ss.fff").
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column or parameter. For example, the display size of a column defined as GRAPHIC(10) is 20.
SQL_LONGVARGRAPHIC	The maximum length of the column or parameter.

Converting Data from SQL to C Data Types

For a given SQL data type:

- the first column of the table lists the legal input values of the *fCType* argument in `SQLBindCol()` and `SQLGetData()`.
- the second column lists the outcomes of a test, often using the *cbValueMax* argument specified in `SQLBindCol()` or `SQLGetData()`, which the driver performs to determine if it can convert the data.
- the third and fourth columns list the values (for each outcome) of the *rgbValue* and *pcbValue* arguments specified in the `SQLBindCol()` or `SQLGetData()` after the driver has attempted to convert the data.
- the last column lists the SQLSTATE returned for each outcome by `SQLFetch()`, `SQLExtendedFetch()`, `SQLGetData()` or `SQLGetSubString()`.

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

If the *fCType* argument in SQLBindCol() or SQLGetData() contains a value not shown in the table for a given SQL data type, SQLFetch(), or SQLGetData() returns the SQLSTATE 07006 (Restricted data type attribute violation).

If the *fCType* argument contains a value shown in the table but which specifies a conversion not supported by the driver, SQLFetch(), or SQLGetData() returns SQLSTATE HYC00 (Driver not capable).

Though it is not shown in the tables, the *pcbValue* argument contains SQL_NULL_DATA when the SQL data value is NULL. For an explanation of the use of *pcbValue* when multiple calls are made to retrieve data, see SQLGetData().

When SQL data is converted to character C data, the character count returned in *pcbValue* does not include the null termination byte. If *rgbValue* is a null pointer, SQLBindCol() or SQLGetData() returns SQLSTATE HY009 (Invalid argument value).

In the following tables:

Length of data
: the total length of the data after it has been converted to the specified C data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is returned to the application.

Significant digits
: the minus sign (if needed) and the digits to the left of the decimal point.

Display size
: the total number of bytes needed to display data in the character format.

Converting Character SQL Data to C Data

The character SQL data types are:

- SQL_CHAR
- SQL_VARCHAR
- SQL_LONGVARCHAR
- SQL_CLOB

Table 198. Converting Character SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Length of data < cbValueMax	Data	Length of data	00000
	Length of data >= cbValueMax	Truncated data	Length of data	01004
SQL_C_BINARY	Length of data <= cbValueMax	Data	Length of data	00000
	Length of data > cbValueMax	Truncated data	Length of data	01004
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_FLOAT SQL_C_TINYINT SQL_C_BIT SQL_C_UBIGINT SQL_C_SBIGINT SQL_C_NUMERIC [c]	Data converted without truncation [a]	Data	Size of the C data type	00000
	Data converted with truncation, but without loss of significant digits [a]	Data	Size of the C data type	01004
	Conversion of data would result in loss of significant digits[a]	Untouched	Size of the C data type	22003
	Data is not a number [a]	Untouched	Size of the C data type	22005
SQL_C_DATE	Data value is a valid date [a]	Data	6 [b]	00000
	Data value is not a valid date [a]	Untouched	6 [b]	22007
SQL_C_TIME	Data value is a valid time [a]	Data	6 [b]	00000
	Data value is not a valid time [a]	Untouched	6 [b]	22007
SQL_C_TIMESTAMP	Data value is a valid timestamp [a]	Data	16 [b]	00000
	Data value is not a valid timestamp [a]	Untouched	16 [b]	22007

Note:

a The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

b This is the size of the corresponding C data type.

c SQL_C_NUMERIC is only supported on 32-bit Windows platforms.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Graphic SQL Data to C Data

The graphic SQL data types are:

SQL_GRAPHIC
SQL_VARGRAPHIC
SQL_LONGVARGRAPHIC
SQL_DBCLOB

Table 199. Converting Graphic SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Number of double byte characters * 2 <= cbValueMax	Data	Length of data(octects)	00000
	Number of double byte characters * 2 > cbValueMax	Truncated data, to the nearest even byte that is less than *cbValueMax*.	Length of data(octects)	01004
SQL_C_DBCHAR	Number of double byte characters * 2 < cbValueMax	Data	Length of data(octects)	00000
	Number of double byte characters * 2 >= cbValueMax	Truncated data, to the nearest even byte that is less than *cbValueMax*.	Length of data(octects)	01004

Note: SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

When converting to floating point values, SQLSTATE 22003 will not be returned if non-significant digits of the resulting value are lost.

Converting Numeric SQL Data to C Data

The numeric SQL data types are:

SQL_DECIMAL
SQL_NUMERIC
SQL_SMALLINT
SQL_INTEGER
SQL_BIGINT

SQL_REAL
SQL_FLOAT
SQL_DOUBLE

Table 200. Converting Numeric SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Length of data	00000
	Number of significant digits < cbValueMax	Truncated data	Length of data	01004
	Number of significant digits >= cbValueMax	Untouched	Length of data	22003
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_DOUBLE SQL_C_TINYINT SQL_C_BIT SQL_C_UBIGINT SQL_C_SBIGINT SQL_C_NUMERIC [b]	Data converted without truncation [a]	Data	Size of the C data type	00000
	Data converted with truncation, but without loss of significant digits [a]	Truncated data	Size of the C data type	01004
	Conversion of data would result in loss of significant digits [a]	Untouched	Size of the C data type	22003

Note:

a The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

b SQL_C_NUMERIC is only supported on 32-bit Windows platforms.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Binary SQL Data to C Data

The binary SQL data types are:

SQL_BINARY
SQL_VARBINARY
SQL_LONGVARBINARY
SQL_BLOB

Table 201. Converting Binary SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	(Length of data) < cbValueMax	Data	Length of data	N/A
	(Length of data) >= cbValueMax	Truncated data	Length of data	01004
SQL_C_BINARY	Length of data <= cbValueMax	Data	Length of data	N/A
	Length of data > cbValueMax	Truncated data	Length of data	01004

Converting Date SQL Data to C Data

The date SQL data type is:

SQL_DATE

Table 202. Converting Date SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 11	Data	10	00000
	cbValueMax < 11	Untouched	10	22003
SQL_C_DATE	None [a]	Data	6 [b]	00000
SQL_C_TIMESTAMP	None [a]	Data [c]	16 [b]	00000

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The time fields of the TIMESTAMP_STRUCT structure are set to zero.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

When the date SQL data type is converted to the character C data type, the resulting string is in the "yyyy-mm-dd” format.

Converting Time SQL Data to C Data

The time SQL data type is:

SQL_TIME

Table 203. Converting Time SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 9	Data	8	00000
	cbValueMax < 9	Untouched	8	22003
SQL_C_TIME	None [a]	Data	6 [b]	00000
SQL_C_TIMESTAMP	None [a]	Data [c]	16 [b]	00000

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The date fields of the TIMESTAMP_STRUCT structure are set to the current system date of the machine that the application is running, and the time fraction is set to zero.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

When the time SQL data type is converted to the character C data type, the resulting string is in the "hh:mm:ss" format.

Converting Timestamp SQL Data to C Data

The timestamp SQL data type is:

SQL_TIMESTAMP

Table 204. Converting Timestamp SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Length of data	00000
	19 <= cbValueMax <= Display size	Truncated Data [b]	Length of data	01004
	cbValueMax < 19	Untouched	Length of data	22003
SQL_C_DATE	None [a]	Truncated data [c]	6 [e]	01004
SQL_C_TIME	None [a]	Truncated data [d]	6 [e]	01004
SQL_C_TIMESTAMP	None [a]	Data	16 [e]	00000

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] The fractional seconds of the timestamp are truncated.

[c] The time portion of the timestamp is deleted.

[d] The date portion of the timestamp is deleted.

[e] This is the size of the corresponding C data type.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

When the timestamp SQL data type is converted to the character C data type, the resulting string is in the "yyyy-mm-dd hh:mm:ss[.fff[fff]]" format (regardless of the precision of the timestamp SQL data type).

SQL to C Data Conversion Examples

Table 205. SQL to C Data Conversion Examples

SQL Data Type	SQL Data Value	C Data Type	cbValue Max	rgbValue	SQL STATE
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0 [a]	00000
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0 [a]	00000
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	4	---	22003
SQL_DECIMAL	1234.56	SQL_C_FLOAT	ignored	1234.56	00000

Table 205. SQL to C Data Conversion Examples (continued)

SQL Data Type	SQL Data Value	C Data Type	cbValue Max	rgbValue	SQL STATE
SQL_DECIMAL	1234.56	SQL_C_SHORT	ignored	1234	01004
SQL_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0 [a]	00000
SQL_DATE	1992-12-31	SQL_C_CHAR	10	---	22003
SQL_DATE	1992-12-31	SQL_C_ TIMESTAMP	ignored	1992,12,31, 0,0,0,0 [b]	00000
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 [a]	00000
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 [a]	01004
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	---	22003

Note:

[a] ″\0″ represents a null termination character.

[b] The numbers in this list are the numbers stored in the fields of the TIMESTAMP_STRUCT structure.

SQLSTATE **00**000 is not returned by SQLError(), rather it is indicated when the function returns SQL_SUCCESS.

Converting Data from C to SQL Data Types

For a given C data type:

- the first column of the table lists the legal input values of the *fSqlType* argument in SQLBindParameter() or SQLSetParam().
- the second column lists the outcomes of a test, often using the length of the parameter data as specified in the *pcbValue* argument in SQLBindParameter() or SQLSetParam(), which the driver performs to determine if it can convert the data.
- the third column lists the SQLSTATE returned for each outcome by SQLExecDirect() or SQLExecute().

Note: Data is sent to the data source only if the SQLSTATE is 00000 (Success).

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

If the *fSqlType* argument in SQLBindParameter() or SQLSetParam() contains a value not shown in the table for a given C data type, SQLSTATE 07006 is returned (Restricted data type attribute violation).

If the *fSqlType* argument contains a value shown in the table but which specifies a conversion not supported by the driver, SQLBindParameter() or SQLSetParam() returns SQLSTATE HYC00 (Driver not capable).

If the *rgbValue* and *pcbValue* arguments specified in SQLBindParameter() or SQLSetParam() are both null pointers, that function returns SQLSTATE HY009 (Invalid argument value).

Length of data
the total length of the data after it has been converted to the specified SQL data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is sent to the data source.

Column length
the maximum number of bytes returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte.

Display size
the maximum number of bytes needed to display data in character form.

Significant digits
the minus sign (if needed) and the digits to the left of the decimal point.

Converting Character C Data to SQL Data

The character C data type is:

SQL_C_CHAR

Table 206. Converting Character C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Length of data <= Column length	00000
	Length of data > Column length	22001
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_REAL SQL_FLOAT SQL_DOUBLE	Data converted without truncation	00000
	Data converted with truncation, but without loss of significant digits	22001
	Conversion of data would result in loss of significant digits	22003
	Data value is not a numeric value	22005
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	(Length of data) < Column length	N/A
	(Length of data) >= Column length	22001
	Data value is not a hexadecimal value	22005
SQL_DATE	Data value is a valid date	00000
	Data value is not a valid date	22007
SQL_TIME	Data value is a valid time	00000
	Data value is not a valid time	22007
SQL_TIMESTAMP	Data value is a valid timestamp	00000
	Data value is not a valid timestamp	22007
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC SQL_DBCLOB	Length of data / 2 <= Column length	00000
	Length of data / 2 < Column length	22001

Note: SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Numeric C Data to SQL Data

The numeric C data types are:

SQL_C_SHORT
SQL_C_LONG
SQL_C_FLOAT
SQL_C_DOUBLE
SQL_C_TINYINT
SQL_C_BIT

Table 207. Converting Numeric C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_REAL SQL_FLOAT SQL_DOUBLE	Data converted without truncation	00000
	Data converted with truncation, but without loss of significant digits	22001
	Conversion of data would result in loss of significant digits	22003
SQL_CHAR SQL_VARCHAR	Data converted without truncation.	00000
	Conversion of data would result in loss of significant digits.	22003

Note: SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

When converting to floating point values, SQLSTATE 22003 will not be returned if non-significant digits of the resulting value are lost.

Converting Binary C Data to SQL Data

The binary C data type is:

SQL_C_BINARY

Table 208. Converting Binary C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Length of data <= Column length	N/A
	Length of data > Column length	22001
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Length of data <= Column length	N/A
	Length of data > Column length	22001

Converting DBCHAR C Data to SQL Data

The Double Byte C data type is:

SQL_C_DBCHAR

Table 209. Converting DBCHAR C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Length of data <= Column length x 2	N/A
	Length of data > Column length x 2	22001
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Length of data <= Column length x 2	N/A
	Length of data > Column length x 2	22001

Converting Date C Data to SQL Data

The date C data type is:

SQL_C_DATE

Table 210. Converting Date C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 10	00000
	Column length < 10	22003
SQL_DATE	Data value is a valid date	00000
	Data value is not a valid date	22007
SQL_TIMESTAMP [a]	Data value is a valid date	00000
	Data value is not a valid date	22007

Note: SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Note: a, the time component of TIMESTAMP is set to zero.

Converting Time C Data to SQL Data

The time C data type is:

SQL_C_TIME

Table 211. Converting Time C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 8	00000
	Column length < 8	22003
SQL_TIME	Data value is a valid time	00000
	Data value is not a valid time	22007
SQL_TIMESTAMP [a]	Data value is a valid time	00000
	Data value is not a valid time	22007

Note: SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Note: a The date component of TIMESTAMP is set to the system date of the machine at which the application is running.

Converting Timestamp C Data to SQL Data

The timestamp C data type is:

SQL_C_TIMESTAMP

Table 212. Converting Timestamp C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= Display size	00000
	19 <= Column length < Display size [a]	22001
	Column length < 19	22003
SQL_DATE	Data value is a valid date [b]	22001
	Data value is not a valid date	22007
SQL_TIME	Data value is a valid time [c]	22001
	Data value is not a valid time	22007
SQL_TIMESTAMP	Data value is a valid timestamp	00000
	Data value is not a valid timestamp	22007

Note:

a The fractional seconds of the timestamp are truncated.

b The time portion of the timestamp is deleted.

c The date portion of the timestamp is deleted.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

C to SQL Data Conversion Examples

Table 213. C to SQL Data Conversion Examples

C Data Type	C Data Value	SQL Data Type	Column length	SQL Data Value	SQL STATE
SQL_C_CHAR	abcdef\0	SQL_CHAR	6	abcdef	00000
SQL_C_CHAR	abcdef\0	SQL_CHAR	5	abcde	22001
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6	1234.56	00000
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	5	1234.5	22001
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	3	---	22003
SQL_C_FLOAT	1234.56	SQL_FLOAT	not applicable	1234.56	00000
SQL_C_FLOAT	1234.56	SQL_INTEGER	not applicable	1234	22001

Note: SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Appendix G. Catalog Views for Stored Procedures

The DB2 CLI functions SQLProcedures() and SQLProcedureColumns() make use of the following two catalog tables to retrieve information about stored procedures and their attributes:

- SYSCAT.PROCEDURES
- SYSCAT.PROCPARMS

For more information about these tables and using the CREATE PROCEDURE command to register a stored procedure in this table, refer to *SQL Reference*.

Appendix H. Pseudo Catalog Table for Stored Procedure Registration

For versions of DB2 CLI before DB2 Universal Database, the DB2CLI.PROCEDURES table must be created and populated at the server before `SQLProcedures()` and `SQLProcedureColumns()` can be called to retrieve information about stored procedures and their attributes.

DB2 Universal Database now makes use of the SYSCAT.PROCEDURES and SYSCAT.PROCPARAMS catalog tables which contain information about stored procedures and therefore the DB2CLI.PROCEDURES table is no longer required.

If you are still using a version of DB2 before version 5, you can use the sample command line processor input file `STORPROC.DDL` to create the DB2CLI.PROCEDURES table. You may then modify the sample `STORPROC.XMP` file to insert rows into this PROCEDURES table. Both of these files are located in the `misc` subdirectory of the `sqllib` directory. To use the file to create the table, execute the following from a command line:

```
db2 -f STORPROC.DDL -z STORPROC.LOG -t
```

It is the database administrator's responsibility to ensure that information has been entered correctly into the table and to keep the table up to date. Initially, all users have SELECT privilege for this table and only users with DBADM authority can INSERT, DELETE or UPDATE rows in this table. As with other tables, a user with DBADM authority can grant privileges to other users.

Legend for the DB2CLI.PROCEDURES Table:

Column Name
Name of the column

Data Type
Data type of the column

Nullable

Yes — Nulls are permitted
No — Nulls are not permitted

Key

Key — The column is part of a primary key
No — The column is **not** part of a key

Description
Description of the column

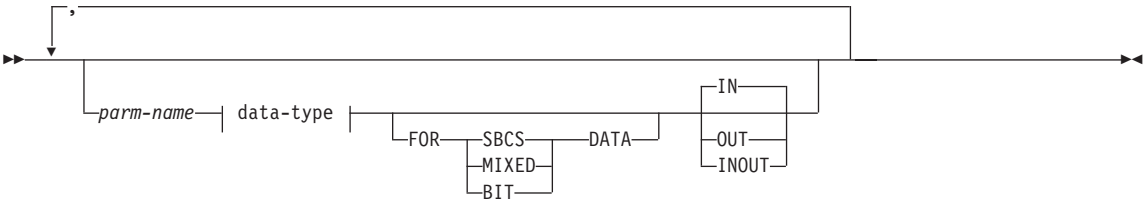
Table 214. Columns of the PROCEDURES table in the DB2CLI schema

Column Name	Data Type	Nullable	Key	Description
PROCSCHEMA	VARCHAR(18)	No	PK	Schema name of the procedure.
PROCNAME	VARCHAR(18)	No	PK	Name of the stored procedure specified on the SQL CALL statement.
DEFINER	VARCHAR(8)	No	No	Definer of the stored procedure. (The database administrator who inserted this row into the table.)
PKGSCHEMA	VARCHAR(18)	No	No	Schema name of the package to be used when the stored procedure is executed.
PKGNAME	VARCHAR(18)	No	No	Name of the package to be loaded when the stored procedure is executed.
PROC_LOCATION	VARCHAR(254)	No	No	External (full path) name of the procedure.
PARM_STYLE	CHAR(1)	No	No	The convention used to pass parameters to the stored procedure: • *D* for the Database Application Remote Interface (DARI) convention used by DB2 for common server servers.
LANGUAGE	CHAR(8)	No	No	The programming language used to create the stored procedure. Possible values are COBOL, C, REXX and FORTRAN for for common servers of DB2. (The value C is used for both C and C++ programs.) Other products may enable other languages, for example: PL/I and BASIC.

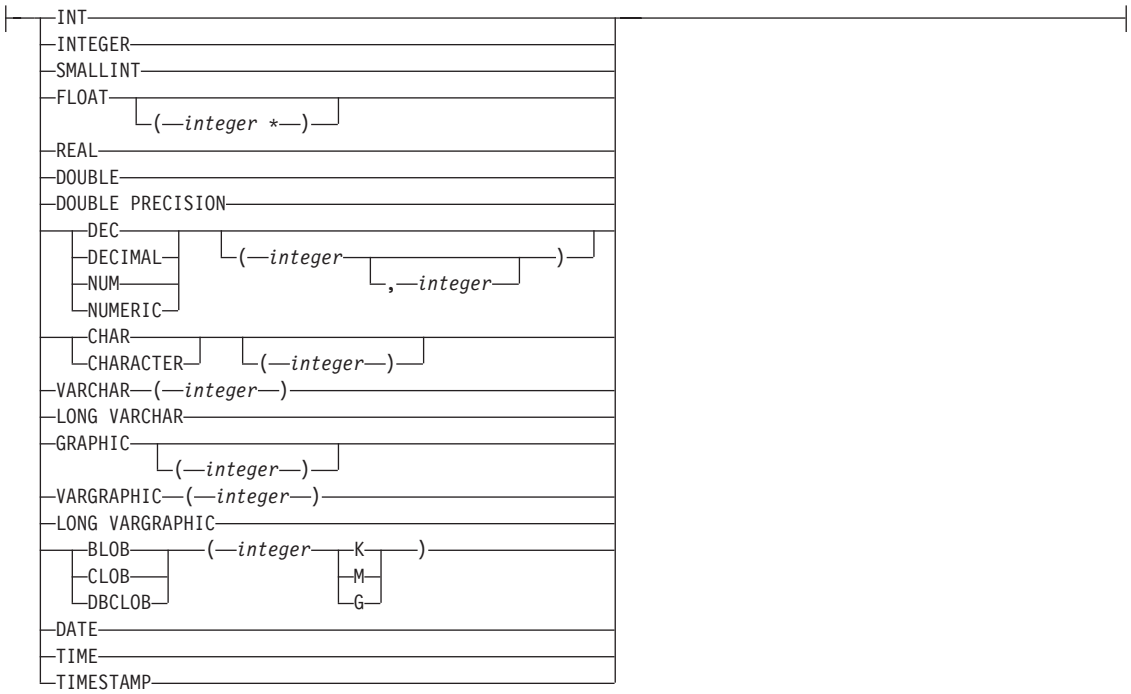
Table 214. Columns of the PROCEDURES table in the DB2CLI schema (continued)

Column Name	Data Type	Nullable	Key	Description
STAYRESIDENT	CHAR(1)	No	No	Determines whether the stored procedure load module is deleted from memory when the stored procedure ends: • *Y* indicates that the load module remains resident in memory after the stored procedure ends. This includes the case when the stored procedure returns SQLZ_HOLD_PROC to stay resident for a number of calls, and then terminates by returning SQLZ_DISCONNECT_PROC. • A *blank* entry indicates that the load module is deleted from memory after the stored module terminates.
RUNOPTS	VARCHAR(254)	No	No	Reserved (empty string).
PARM_LIST	VARCHAR(3000)	No	No	Parameter list of the stored procedure. See the syntax diagram following this table for the format of this parameter list.
FENCED	CHAR(1)	No	No	An indication of whether or not the procedure runs “fenced”: • *Y* indicates the stored procedure is fenced • *N* indicates the stored procedure is not fenced
REMARKS	VARCHAR(254)	Yes	No	Description of the stored procedure.
RESULT_SETS	SMALLINT	No	No	The number of result sets that can be returned.

The input format of the parameter list column, PARM_LIST, is defined in Figure 19 on page 842. If there is a syntax error in the contents of this column, a call to `SQLProcedureColumns()` will result in an error.



data-type



Note: * - not supported in common server versions of DB2

Figure 19. PARMLIST String Syntax. This PARMLIST syntax diagram combines the data types supported for both DB2 for MVS/ESA and for common server versions of DB2.

Appendix I. Supported SQL Statements

Table 215. SQL Statements (DB2 Universal Database)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)
ALTER { BUFFERPOOL, NODEGROUP, TABLE, TABLESPACE, TYPE, VIEW }	X	X	X
BEGIN DECLARE SECTION[2]			
CALL		X[9]	X[4]
CLOSE		X	SQLCloseCursor(), SQLFreeStmt()
COMMENT ON	X	X	X
COMMIT	X	X	SQLEndTran, SQLTransact()
Compound SQL			X[4]
CONNECT (Type 1)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()
CONNECT (Type 2)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()
CREATE { ALIAS, BUFFERPOOL, DISTINCT TYPE, EVENT MONITOR, FUNCTION, INDEX, NODEGROUP, PROCEDURE, SCHEMA, TABLE, TABLESPACE, TRIGGER, TYPE, VIEW }	X	X	X
DECLARE CURSOR[2]		X	SQLAllocStmt()
DELETE	X	X	X
DESCRIBE[8]		X	SQLColAttributes(), SQLDescribeCol(), SQLDescribParam()[6]
DISCONNECT		X	SQLDisconnect()
DROP	X	X	X
END DECLARE SECTION[2]			
EXECUTE			SQLExecute()
EXECUTE IMMEDIATE			SQLExecDirect()
EXPLAIN	X	X	X

Table 215. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)
FETCH		X	SQLExtendedFetch()[7], SQLFetch(), SQLFetchScroll()[7]
FREE LOCATOR			X[4]
GRANT	X	X	X
INCLUDE[2]			
INSERT	X	X	X
LOCK TABLE	X	X	X
OPEN		X	SQLExecute(), SQLExecDirect()
PREPARE			SQLPrepare()
RELEASE		X	
RENAME TABLE	X	X	X
REVOKE	X	X	X
ROLLBACK	X	X	SQLEndTran(), SQLTransact()
select-statement	X	X	X
SELECT INTO			
SET CONNECTION		X	SQLSetConnection()
SET CONSTRAINTS	X	X	X
SET CURRENT DEGREE	X	X	X
SET CURRENT EXPLAIN MODE	X	X	X, SQLSetConnectAttr()
SET CURRENT EXPLAIN SNAPSHOT	X	X	X, SQLSetConnectAttr()
SET CURRENT FUNCTION PATH	X	X	X
SET CURRENT PACKAGESET			
SET CURRENT QUERY OPTIMIZATION	X	X	X
SET EVENT MONITOR STATE	X	X	X
SET transition-variable[5]	X	X	X
SIGNAL SQLSTATE[5]	X	X	X
UPDATE	X	X	X
VALUES INTO			
WHENEVER[2]			

Table 215. SQL Statements (DB2 Universal Database) (continued)

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

Note:

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. For information on the DESCRIBE SQL statement, refer to the *SQL Reference*. For information on the DESCRIBE CLP command, refer to the *Command Reference*.
9. When CALL is issued through the command line processor, only certain procedures and their respective parameters are supported .

Appendix J. CLI Sample Code

This appendix lists all the sample DB2 CLI programs, including those demonstrated in this book. The samples can be found on OS/2 and Windows 32-bit operating systems in `sqllib\samples\cli`, and on UNIX platforms in `sqllib/samples/cli`. The `README` file contains a complete description of each file, and describes how to use the `makefile` and build scripts to build all the samples.

The following table lists all the DB2 CLI sample programs.

Table 216. Sample CLI Programs in DB2 Universal Database

Sample Program Name	Program Description
Utility files used by most CLI samples	
`samputil.c`	Utility functions used by most samples
`samputil.h`	Header file for `samputil.c`, included by most samples
General CLI Samples	
`adhoc.c`	Interactive SQL with formatted output (was `typical.c`)
`async.c`	Run a function asynchronously (based on `fetch.c`)
`basiccon.c`	Basic connection
`browser.c`	List columns, foreign keys, index columns or stats for a table
`calludf.c`	Register and call a UDF
`colpriv.c`	List column Privileges
`columns.c`	List all columns for table search string
`compnd.c`	Compound SQL example
`datasour.c`	List all available data sources
`descrptr.c`	Example of descriptor usage
`drivrcon.c`	Rewrite of `basiccon.c` using SQLDriverConnect
`duowcon.c`	Multiple DUOW Connect type 2, syncpoint 1 (one phase commit)
`embedded.c`	Show equivalent DB2 CLI calls, for embedded SQL (in comments)
`fetch.c`	Simple example of a fetch sequence
`getattrs.c`	List some common environment, connection and statement options/attributes
`getcurs.c`	Show use of SQLGetCursor, and positioned update
`getdata.c`	Rewrite of `fetch.c` using SQLGetData instead of SQLBindCol
`getfuncs.c`	List all supported functions

Table 216. Sample CLI Programs in DB2 Universal Database (continued)

Sample Program Name	Program Description
getfuncs.h	Header file for getfuncs.c
getinfo.c	Use SQLGetInfo to get driver version and other information
getsqlca.c	Rewrite of adhoc.c to use prepare/execute and show cost estimate
lookres.c	Extract string from the resume clob field using locators
mixed.sqc	CLI sample with functions written using embedded SQL (Note: This file must be precompiled)
multicon.c	Multiple connections
native.c	Simple example of calling SQLNativeSql, and SQLNumParams
prepare.c	Rewrite of fetch.c, using prepare/execute instead of execdirect
proccols.c	List procedure parameters using SQLProcedureColumns
procs.c	List procedures using SQLProcedures
sfetch.c	Scrollable cursor example (based on xfetch.c)
setcolat.c	Set column attributes (using SQLSetColAttributes)
setcurs.c	Rewrite of getcurs.c using SQLSetCurs for positioned update
seteattr.c	Set environment attribute (SQL_ATTR_OUTPUT_NTS)
tables.c	List all tables
typeinfo.c	Display type information for all types for current data source
xfetch.c	Extended Fetch, multiple rows per fetch
BLOB Samples	
picin.c	Loads graphic BLOBS into the emp_photo table directly from a file using SQLBindParamToFile
picin2.c	Loads graphic BLOBS into the emp_photo table using SQLPutData
showpic.c	Extracts BLOB picture to file (using SQLBindColToFile), then displays the graphic.
showpic2.c	Extracts BLOB picture to file using piecewise output, then displays the graphic.
Stored Procedure Samples	
clicall.c	Defines a CLI function which is used in the embedded SQL sample mrspcli3.sqc
inpcli.c	Call embedded input stored procedure samples/c/inpsrv
inpcli2.c	Call CLI input stored procedure inpsrv2
inpsrv2.c	CLI input stored procedure (rewrite of embedded sample inpsrv.sqc)
mrspcli.c	CLI program that calls mrspsrv.c
mrspcli2.c	CLI program that calls mrspsrv2.sqc
mrspcli3.sqc	An embedded SQL program that calls mrspsrv2.sqc using clicall.c

Table 216. Sample CLI Programs in DB2 Universal Database (continued)

Sample Program Name	Program Description
mrspsrv.c	Stored procedure that returns a multi-row result set
mrspsrv2.sqc	An embedded SQL stored procedure that returns a multi-row result set
outcli.c	Call embedded output stored procedure samples/c/inpsrv
outcli2.c	Call CLI output stored procedure inpsrv2
outsrv2.c	CLI output stored procedure (rewrite of embedded sample inpsrv.sqc)
Samples using ORDER tables created by create.c (Run in the following order)	
create.c	Creates all tables for the order scenario
custin.c	Inserts customers into the customer table (array insert)
prodin.c	Inserts products into the products table (array insert)
prodpart.c	Inserts parts into the prod_parts table (array insert)
ordin.c	Inserts orders into the ord_line, ord_cust tables (array insert)
ordrep.c	Generates order report using multiple result sets
partrep.c	Generates exploding parts report (recursive SQL Query)
order.c	UDF library code (declares a 'price' UDF)
order.exp	Used to build order libary
Samples unchanged from DB2 Version 2	
v2sutil.c	samputil.c using old v2 functions
v2sutil.h	samputil.h using old v2 functions
v2fetch.c	fetch.c using old v2 functions
v2xfetch.c	xfetch.c using old v2 functions

Note: Other files in the samples/cli directory include:

- README - Lists all example files.
- makefile - Makefile for all files
- build files for applications and stored procedures

The following two example files are listed in this section.:

- embedded.c - compares DB2 CLI and embedded calls
- adhoc.c - a full interactive SQL example

Embedded SQL Example

This example is a modified version of the example contained in the X/Open SQL CLI document. It shows embedded statements in comments, and the equivalent DB2 CLI function calls.

```
/* From CLI sample embedded.c */
/* ... */
#include <string.h>
#include <stdlib.h>
#include <sqlcli1.h>
#include "samputil.h"          /* Header file for CLI sample code */

/* ... */

/*
 Global Variables for user id and password.
 To keep samples simple, not a recommended practice.
*/
extern SQLCHAR server[SQL_MAX_DSN_LENGTH + 1] ;
extern SQLCHAR uid[MAX_UID_LENGTH + 1] ;
extern SQLCHAR pwd[MAX_PWD_LENGTH + 1] ;

int main( int argc, char * argv[] ) {

    SQLHANDLE henv, hdbc, hstmt ;
    SQLRETURN rc ;

    SQLINTEGER id ;
    SQLCHAR name[51] ;

    SQLCHAR * create = "CREATE TABLE NAMEID (ID integer, NAME varchar(50))" ;
    SQLCHAR * insert = "INSERT INTO NAMEID VALUES (?, ?)" ;
    SQLCHAR * select = "select ID, NAME from NAMEID" ;
    SQLCHAR * drop   = "DROP TABLE NAMEID" ;

/* ... */

    /* EXEC SQL CONNECT TO :server USER :uid USING :authentication_string; */
    /* macro to initalize server, uid and pwd */
    INIT_UID_PWD ;
    /* allocate an environment handle */
    rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;
    /* allocate a connect handle, and connect */
    rc = DBconnect( henv, &hdbc ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;
    /* allocate a statement handle */
    rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

    /* EXEC SQL CREATE TABLE NAMEID (ID integer, NAME varchar(50)); */
    /* execute the sql statement */
    rc = SQLExecDirect( hstmt, create, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* EXEC SQL COMMIT WORK; */
    /* commit create table */
    rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;
```

```
/* EXEC SQL INSERT INTO NAMEID VALUES ( :id, :name ); */
/* show the use of SQLPrepare/SQLExecute method */
/* prepare the insert */
rc = SQLPrepare( hstmt, insert, SQL_NTS ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
/* Set up the first input parameter "id" */
rc = SQLBindParameter( hstmt,
                       1,
                       SQL_PARAM_INPUT,
                       SQL_C_LONG,
                       SQL_INTEGER,
                       0,
                       0,
                       (SQLPOINTER) & id,
                       0,
                       NULL
                     ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
/* Set up the second input parameter "name" */
rc = SQLBindParameter( hstmt,
                       2,
                       SQL_PARAM_INPUT,
                       SQL_C_CHAR,
                       SQL_VARCHAR,
                       51,
                       0,
                       name,
                       51,
                       NULL
                     ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
/* now assign parameter values and execute the insert */
id = 500 ;
strcpy( name, "Babbage" ) ;
rc = SQLExecute( hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* EXEC SQL COMMIT WORK; */
/* commit inserts */
rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;
/*  Reset input parameter.  */
rc = SQLFreeStmt( hstmt, SQL_RESET_PARAMS ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* EXEC SQL DECLARE c1 CURSOR FOR SELECT ID, NAME FROM NAMEID; */
/* EXEC SQL OPEN c1; */
/* The application doesn't specify "declare c1 cursor for" */
rc = SQLExecDirect( hstmt, select, SQL_NTS ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* EXEC SQL FETCH c1 INTO :id, :name; */
/* Binding first column to output variable "id" */
SQLBindCol( hstmt, 1, SQL_C_LONG, ( SQLPOINTER ) & id, 0, NULL ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

```
    /* Binding second column to output variable "name" */
    SQLBindCol( hstmt, 2, SQL_C_CHAR, name, 51, NULL ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
    /* now execute the fetch        */
    while ( ( rc = SQLFetch( hstmt ) ) == SQL_SUCCESS )
       printf( "Result of Select: id = %ld name = %s\n", id, name ) ;
    if (rc != SQL_NO_DATA_FOUND)
       CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* finally, we should commit, discard hstmt, disconnect */

    /* EXEC SQL COMMIT WORK; */
    /*  Close cursor and free bound columns.  */
    /*  Free statement resources  */
    rc = SQLFreeStmt( hstmt, SQL_UNBIND ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
    rc = SQLFreeStmt( hstmt, SQL_CLOSE ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
    /*  Drop table.  */
    rc = SQLExecDirect( hstmt, drop, SQL_NTS ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
    /* commit the transaction */
    rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

    /* EXEC SQL CLOSE c1; */
    /* free the statement handle */
    rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* EXEC SQL DISCONNECT; */
    /* disconnect from the database */
    printf( "\n>Disconnecting .....\n" ) ;
    rc = SQLDisconnect( hdbc ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;
    /* free the connection handle */
    rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc ) ;
    CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;
    /* free the environment handle */
    rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    return( SQL_SUCCESS ) ;

}
```

Interactive SQL Example

This example is a modified version of the example contained in the X/Open SQL CLI document. It shows the execution of interactive SQL statements, and follows the flow described in “Chapter 2. Writing a DB2 CLI Application” on page 9.

```
/* From CLI sample adhoc.c */
/* ... */
/******************************************************************
** process_stmt
** - allocates a statement resources
** - executes the statement
** - determines the type of statement
**   - if there are no result columns, therefore non-select statement
**      - if rowcount > 0, assume statement was UPDATE, INSERT, DELETE
**     else
**      - assume a DDL, or Grant/Revoke statement
**   else
**      - must be a select statement.
**      - display results
** - frees the statement resources
******************************************************************/

int process_stmt( SQLHANDLE hstmt, SQLCHAR * sqlstr ) {

    SQLSMALLINT     nresultcols;
    SQLINTEGER      rowcount;
    SQLRETURN       rc;

    /* execute the SQL statement in "sqlstr"    */

    rc = SQLExecDirect(hstmt, sqlstr, SQL_NTS);
    if (rc != SQL_SUCCESS)
        if (rc == SQL_NO_DATA_FOUND) {
            printf("\nStatement executed without error, however,\n");
            printf("no data was found or modified\n");
            return (SQL_SUCCESS);
        }
        else CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLNumResultCols(hstmt, &nresultcols);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    /* determine statement type */
    if (nresultcols == 0) {     /* statement is not a select statement */
        rc = SQLRowCount(hstmt, &rowcount);
        if (rowcount > 0) /* assume statement is UPDATE, INSERT, DELETE */
            printf("Statement executed, %ld rows affected\n", rowcount);
        else /* assume statement is GRANT, REVOKE or a DLL statement */
           printf( "Statement completed successful\n" ) ;
    }
    else print_results( hstmt ) ; /* display the result set */
    /* end determine statement type */

    /* free statement resources */

    rc = SQLFreeStmt( hstmt, SQL_UNBIND ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLFreeStmt( hstmt, SQL_RESET_PARAMS ) ;
```

```
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    rc = SQLFreeStmt( hstmt, SQL_CLOSE ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

    return( 0 ) ;

}                           /* end process_stmt */

/* From CLI sample samputil.c */
/* ... */
/* print_results */

SQLRETURN print_results( SQLHANDLE hstmt ) {

    SQLCHAR     colname[32] ;
    SQLSMALLINT coltype ;
    SQLSMALLINT colnamelen ;
    SQLSMALLINT nullable ;
    SQLUINTEGER collen[MAXCOLS] ;
    SQLSMALLINT scale ;
    SQLINTEGER  outlen[MAXCOLS] ;
    SQLCHAR *   data[MAXCOLS] ;
    SQLCHAR     errmsg[256] ;
    SQLRETURN   rc ;
    SQLSMALLINT nresultcols, i ;
    SQLINTEGER  displaysize ;

    rc = SQLNumResultCols( hstmt, &nresultcols ) ;
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
    for ( i = 0; i < nresultcols; i++ ) {
        SQLDescribeCol( hstmt,
                        ( SQLSMALLINT ) ( i + 1 ),
                        colname,
                        sizeof(colname),
                        &colnamelen,
                        &coltype,
                        &collen[i],
                        &scale,
                        NULL
                      ) ;
        /* get display length for column */
        SQLColAttribute( hstmt,
                         ( SQLSMALLINT ) ( i + 1 ),
                         SQL_DESC_DISPLAY_SIZE,
                         NULL,
                         0,
                         NULL,
                         &displaysize
                       ) ;

        /*
         Set column length to max of display length,
         and column name length. Plus one byte for
         null terminator.
```

```
    */
    collen[i] = max( displaysize,
                     strlen( ( char * ) colname )
                   ) + 1 ;

    printf( "%-*.*s",
            ( int ) collen[i],
            ( int ) collen[i],
            colname
          ) ;

    /* allocate memory to bind column */
    data[i] = ( SQLCHAR * ) malloc( ( int ) collen[i] ) ;

    /* bind columns to program vars, converting all types to CHAR */
    SQLBindCol( hstmt,
                ( SQLSMALLINT ) ( i + 1 ),
                SQL_C_CHAR,
                data[i],
                collen[i],
                &outlen[i]
              ) ;
}

printf( "\n" ) ;
/* display result rows */
while ( SQLFetch( hstmt ) != SQL_NO_DATA ) {
    errmsg[0] = '\0' ;
    for ( i = 0; i < nresultcols; i++ ) {
        /* Check for NULL data */
        if ( outlen[i] == SQL_NULL_DATA )
           printf( "%-*.*s",
                   ( int ) collen[i],
                   ( int ) collen[i],
                   "NULL"
                 ) ;
        else { /* Build a truncation message for any columns truncated */
           if ( outlen[i] >= collen[i] ) {
              sprintf( ( char * ) errmsg + strlen( ( char * ) errmsg ),
                       "%d chars truncated, col %d\n",
                       ( int ) outlen[i] - collen[i] + 1,
                       i + 1
                     ) ;
           }
           /* Print column */
           printf( "%-*.*s",
                   ( int ) collen[i],
                   ( int ) collen[i],
                   data[i]
                 ) ;
        }
    }                          /* for all columns in this row  */

    printf( "\n%s", errmsg ) ; /* print any truncation messages */
}                              /* while rows to fetch */
```

```
    /* free data buffers */
    for ( i = 0; i < nresultcols; i++ ) {
        free( data[i] ) ;
    }

    return( SQL_SUCCESS ) ;

}                               /* end print_results */
```

Appendix K. Using the DB2 CLI/ODBC Trace Facility

Applications using either DB2 CLI or ODBC and the IBM DB2 CLI/ODBC Driver can have all the function calls traced to a plain text file. This can help with problem determination, datababase and application tuning or just to better understand what a 3rd party application is doing.

The trace can be enabled at runtime using the CLI/ODBC Settings notebook, accessible from the Client Configuration Assistant if it is available, or by editing the db2cli.ini file directly. A DB2 CLI application can also enable the trace by setting the SQL_ATTR_TRACE and SQL_ATTR_TRACEFILE Environment or Connection attributes. These are the same attributes used by the tracing facility in the Microsoft ODBC Driver Manager.

Enabling the Trace Using the db2cli.ini File

The db2cli.ini file is located by default in the \sqllib\ path for Intel platforms, and the /sqllib/cfg/ path for UNIX platforms. The DB2 CLI/ODBC configuration keywords used by the trace facility are:

- "TRACE" on page 201
- "TRACEFILENAME" on page 203
- "TRACEPATHNAME" on page 205
- "TRACEFLUSH" on page 204
- "TRACECOMM" on page 202

The following lines must be added to enable the trace (the keywords are NOT case sensitive):

1. [COMMON]
2. Trace=1
3. TraceFileName=(fully qualified filename)

   or

   TracePathname=(fully qualified pathname)
4. TraceFlush=(0 or 1) - optional
5. TraceComm=(0 or 1) - optional

For example:

```
[COMMON]
trace=1
tracefilename=d:\temp\clitrace.txt
```

Setting TRACE to 0 turns tracing off. The trace file information can be left in the configuration file for the next time it is needed. See “TRACE” on page 201 for more information.

If the application does not exit or exits abnormally, the trace file will probably not be complete. Setting TRAEFLUSH to 1 will cause a flush to disk on every function call (which will increase the overhead of tracing dramatically). See “TRACEFLUSH” on page 204 for more information.

To have information about each network request included in the trace file, set TRACECOMM to 1. See “TRACECOMM” on page 202 for more information.

Locating the Resulting Files

If you used a fully qualified filename with the TRACEFILENAME keyword, you should have no problem locating the file. If you used a relative pathname, it will depend on what the operating system considers the current path of the application.

If you used a pathname instead of a filename with the TRACEPATHNAME keyword, you will need to check the directory for a set of files created with the name set to the process id of the application and an extension that is a sequence number for each unique thread (eg. 65397.0, 65397.1, 65397.2 etc.). The file date and timestamp can be used to help locate the relevant file.

If you used a relative pathname, it will depend on what the operating system considers the current path of the application.

If there is no output file:

- Verify that the keywords are set correctly in db2cli.ini.
- Ensure the application is restarted (specifically, `SQLAllocEnv()` must be called to read the db2cli.ini file and initialize the trace).
- Ensure the application has write access to the specified filename.
- Check if the DB2CLIINIPATH environment variable is specified. This environment variable changes the location from which the db2cli.ini file will be read.
- ODBC applications will not access the IBM DB2 CLI/ODBC Driver until the first connect call. No trace entires will be written to the file until the application makes this connect call. See “ODBC Driver Manager Tracing” on page 865 for more details.

Reading the Trace Information

The purpose of the trace is to display the sequence of calls, the input and output arguments and the return code for each function called. The trace is intended for people familiar (or looking to become familiar) with the DB2 CLI or ODBC function calls. Two things that are useful for anyone is the the SQL statement text being executed and any error messages that the application may not be reporting.

To locate:

- SQL statements:

  Search the trace file for the strings ″SQLExecDirect″ and ″SQLPrepare″, you will find the SQL Statement on the same line that contains the text and the ″—>″ input arrow (although your editor may wrap the line).

- Errors: (queried by the application)

  Search the trace file for ″SQLError″, the message text will be shown on the line that contains the string and the output arrow ″<—″.

- Errors: (Ignored by the application)

  Search for ″Unretrieved error message=″ This indicates that a previous call got an SQL_ERROR or SQL_SUCCESS_WITH_INFO return code, but that the application did not query for the error information.

  Note: An application may expect some error messages, you should look at all the error messages in the trace file and try to determine the serious ones.

Detailed Trace File Format

Refer to the example trace file below. Note that the line numbers have been added for this discussion, and do NOT appear in the trace.

- Line 1: The build date and product signature is shown to aid IBM Service.
- Lines 2-3: First of a common two line sequence showing the arguments on input (—>) to the function call. Integer arguments may be mapped to a defined value like ″SQL_HANDLE_ENV″, output arguments are usually shown as pointers, with an ″&″ prefix.
- Lines 4-5: Two line sequence showing ouput (<—) results of the function call. Only output arguments are shown, and the return code on the second line following the (—>). Match this with the preceding input lines.
- Line 7: Example of an elapsed time on input. This is the time in the application between CLI function calls, shown in seconds. (Note: the granularity or accuracy of these timmings vary between platforms).
- Line 8: Example of an elasped time on output. This is the time in DB2 CLI spent executing the function.

- Lines 18-20: Both SQLDriverConnect() and SQLConnect() display the keywords set on both the input connection string and set in the db2cli.ini file.
- Line 23: The output statement handle is shown as 1:1, the first number represents the connection handle, the second the statement handle on that connection. This also applies to descriptor handles, but not to connection or environment handles, where the first number is always zero.
- Line 29: Example of SQL statement text for SQLPrepare().
- Line 43-44: Deferred arguments from SQLBindParameter() calls (lines 33 - 40). This is the data sent for each of the sql parameter markers (?) in the prepared statement (line 29).
- Lines 79-81: The ouptut from the SQLFetch() call. (iCol = Column, rgbValue = data in char format, pcbValue=Length).
- Line 110: SQLError() output, showing message text. The pfNativeError is either the DB2 SQLCODE or -9999 if the error originated from DB2 CLI instead of the database server.
- Line 123: Shows an unretrived error message. This is shown whenever a function is called using a handle which had a previous error, but was never retrieved by the application. It is effectively ″lost″ (to the application) at this point but is captured in the trace.

Example Trace File

This example has line numbers added to aid the discussion, line numbers do NOT appear in the trace.

```
 1  Build Date: 97/05/13-Product: QDB2/6000 (4) - Driver Version: 05.00.0000

 2  SQLAllocHandle( fHandleType=SQL_HANDLE_ENV, hInput=0:0,
          phOutput=&2ff7f388 )
 3       ---> Time elapsed - +1.399700E-002 seconds

 4  SQLAllocHandle( phOutput=0:1 )
 5       <--- SQL_SUCCESS   Time elapsed - +6.590000E-003 seconds

 6  SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=0:1,
          phOutput=&2ff7f378 )
 7       ---> Time elapsed - +1.120000E-002 seconds

 8  SQLAllocHandle( phOutput=0:1 )
 9       <--- SQL_SUCCESS   Time elapsed - +8.979000E-003 seconds

10  SQLSetConnectOption( hDbc=0:1, fOption=SQL_ATTR_AUTOCOMMIT, vParam=0 )
11       ---> Time elapsed - +6.638000E-003 seconds

12  SQLSetConnectOption( )
13       <--- SQL_SUCCESS   Time elapsed - +1.209000E-003 seconds

14  SQLDriverConnect( hDbc=0:1, hwnd=0:0, szConnStrIn="DSN=loopback;
          uid=clitest1;pwd=*******", cbConnStrIn=-3, szConnStrOut=&2ff7e7b4,
```

```
          cbConnStrOutMax=250, pcbConnStrOut=&2ff7e7ae,
          fDriverCompletion=SQL_DRIVER_NOPROMPT )
15        ---> Time elapsed - +1.382000E-003 seconds

16  SQLDriverConnect( szConnStrOut="DSN=LOOPBACK;UID=clitest1;PWD=*******;",
          pcbConnStrOut=38 )
17        <--- SQL_SUCCESS   Time elapsed - +7.675910E-001 seconds
18  ( DSN="LOOPBACK" )
19  ( UID="clitest1" )
20  ( PWD="*******" )

21  SQLAllocHandle( fHandleType=SQL_HANDLE_STMT, hInput=0:1,
          phOutput=&2ff7f378 )
22        ---> Time elapsed - +1.459900E-002 seconds

23  SQLAllocHandle( phOutput=1:1 )
24        <--- SQL_SUCCESS   Time elapsed - +7.008300E-002 seconds

25  SQLExecDirect( hStmt=1:1, pszSqlStr="create table test(id integer,
          name char(20), created date)", cbSqlStr=-3 )
26        ---> Time elapsed - +1.576899E-002 seconds

27  SQLExecDirect( )
28        <--- SQL_SUCCESS   Time elapsed - +1.017835E+000 seconds

29  SQLPrepare( hStmt=1:1, pszSqlStr="insert into test
          values (?, ?, current date)", cbSqlStr=-3 )
30        ---> Time elapsed - +5.008000E-003 seconds

31  SQLPrepare( )
32        <--- SQL_SUCCESS   Time elapsed - +7.896000E-003 seconds

33  SQLBindParameter( hStmt=1:1, iPar=1, fParamType=SQL_PARAM_INPUT,
          fCType=SQL_C_LONG, fSQLType=SQL_INTEGER, cbColDef=4, ibScale=0,
          rgbValue=&20714d88, cbValueMax=4, pcbValue=&20714d54 )
34        ---> Time elapsed - +2.870000E-003 seconds

35  SQLBindParameter( )
36        <--- SQL_SUCCESS   Time elapsed - +3.803000E-003 seconds

37  SQLBindParameter( hStmt=1:1, iPar=2, fParamType=SQL_PARAM_INPUT,
          fCType=SQL_C_CHAR, fSQLType=SQL_CHAR, cbColDef=20, ibScale=0,
          rgbValue=&20714dd8, cbValueMax=21, pcbValue=&20714da4 )
38        ---> Time elapsed - +2.649000E-003 seconds

39  SQLBindParameter( )
40        <--- SQL_SUCCESS   Time elapsed - +3.882000E-003 seconds

41  SQLExecute( hStmt=1:1 )
42        ---> Time elapsed - +3.681000E-003 seconds
43  ( iPar=1, fCType=SQL_C_LONG, rgbValue=10, pcbValue=4, piIndicatorPtr=4 )
44  ( iPar=2, fCType=SQL_C_CHAR, rgbValue="-3", pcbValue=2, piIndicatorPtr=2 )

45  SQLExecute( )
46        <--- SQL_SUCCESS   Time elapsed - +4.273490E-001 seconds
```

```
47  SQLExecute( hStmt=1:1 )
48      ---> Time elapsed - +5.483000E-003 seconds
49  ( iPar=1, fCType=SQL_C_LONG, rgbValue=10, pcbValue=4, piIndicatorPtr=4 )
50  ( iPar=2, fCType=SQL_C_CHAR, rgbValue="-3", pcbValue=2, piIndicatorPtr=2 )

51  SQLExecute( )
52      <--- SQL_SUCCESS   Time elapsed - +1.299300E-002 seconds

53  SQLExecute( hStmt=1:1 )
54      ---> Time elapsed - +3.702000E-003 seconds
55  ( iPar=1, fCType=SQL_C_LONG, rgbValue=10, pcbValue=4, piIndicatorPtr=4 )
56  ( iPar=2, fCType=SQL_C_CHAR, rgbValue="-3", pcbValue=2, piIndicatorPtr=2 )

57  SQLExecute( )
58      <--- SQL_SUCCESS   Time elapsed - +1.265700E-002 seconds

59  SQLExecDirect( hStmt=1:1, pszSqlStr="select * from test", cbSqlStr=-3 )
60      ---> Time elapsed - +2.983000E-003 seconds

61  SQLExecDirect( )
62      <--- SQL_SUCCESS   Time elapsed - +2.469180E-001 seconds

63  SQLBindCol( hStmt=1:1, iCol=1, fCType=SQL_C_LONG, rgbValue=&20714e38,
          cbValueMax=4, pcbValue=&20714e04 )
64      ---> Time elapsed - +5.069000E-003 seconds

65  SQLBindCol( )
66      <--- SQL_SUCCESS   Time elapsed - +2.660000E-003 seconds

67  SQLBindCol( hStmt=1:1, iCol=2, fCType=SQL_C_CHAR, rgbValue=&20714e88,
          cbValueMax=21, pcbValue=&20714e54 )
68      ---> Time elapsed - +2.492000E-003 seconds

69  SQLBindCol( )
70      <--- SQL_SUCCESS   Time elapsed - +2.795000E-003 seconds

71  SQLBindCol( hStmt=1:1, iCol=3, fCType=SQL_C_CHAR, rgbValue=&20714ee8,
          cbValueMax=21, pcbValue=&20714eb4 )
72      ---> Time elapsed - +2.490000E-003 seconds

73  SQLBindCol( )
74      <--- SQL_SUCCESS   Time elapsed - +2.749000E-003 seconds

75  SQLFetch( hStmt=1:1 )
76      ---> Time elapsed - +2.660000E-003 seconds

77  SQLFetch( )
78      <--- SQL_SUCCESS   Time elapsed - +9.200000E-003 seconds
79  ( iCol=1, fCType=SQL_C_LONG, rgbValue=10, pcbValue=4 )
80  ( iCol=2, fCType=SQL_C_CHAR, rgbValue="-3                  ",
      pcbValue=20 )
81  ( iCol=3, fCType=SQL_C_CHAR, rgbValue="1997-05-23", pcbValue=10 )

82  SQLFetch( hStmt=1:1 )
```

```
 83      ---> Time elapsed - +4.942000E-003 seconds

 84  SQLFetch( )
 85      <--- SQL_SUCCESS   Time elapsed - +7.860000E-003 seconds
 86  ( iCol=1, fCType=SQL_C_LONG, rgbValue=10, pcbValue=4 )
 87  ( iCol=2, fCType=SQL_C_CHAR, rgbValue="-3                  ",
         pcbValue=20 )
 88  ( iCol=3, fCType=SQL_C_CHAR, rgbValue="1997-05-23", pcbValue=10 )

 89  SQLFetch( hStmt=1:1 )
 90      ---> Time elapsed - +4.872000E-003 seconds

 91  SQLFetch( )
 92      <--- SQL_SUCCESS   Time elapsed - +7.669000E-003 seconds
 93  ( iCol=1, fCType=SQL_C_LONG, rgbValue=10, pcbValue=4 )
 94  ( iCol=2, fCType=SQL_C_CHAR, rgbValue="-3                  ",
         pcbValue=20 )
 95  ( iCol=3, fCType=SQL_C_CHAR, rgbValue="1997-05-23", pcbValue=10 )

 96  SQLFetch( hStmt=1:1 )
 97      ---> Time elapsed - +5.103000E-003 seconds

 98  SQLFetch( )
 99      <--- SQL_NO_DATA_FOUND   Time elapsed - +6.044000E-003 seconds

100  SQLCloseCursor( hStmt=1:1 )
101      ---> Time elapsed - +2.682000E-003 seconds

102  SQLCloseCursor( )
103      <--- SQL_SUCCESS   Time elapsed - +6.794000E-003 seconds

104  SQLExecDirect( hStmt=1:1, pszSqlStr="select * foo bad sql", cbSqlStr=-3 )
105      ---> Time elapsed - +2.967000E-003 seconds

106  SQLExecDirect( )
107      <--- SQL_ERROR   Time elapsed - +1.103700E-001 seconds

108  SQLError( hEnv=0:0, hDbc=0:0, hStmt=1:1, pszSqlState=&2ff6f19c,
           pfNativeError=&2ff6ed00, pszErrorMsg=&2ff6ed9c, cbErrorMsgMax=1024,
           pcbErrorMsg=&2ff6ed0a )
109      ---> Time elapsed - +2.267000E-003 seconds

110  SQLError( pszSqlState="42601", pfNativeError=-104,
           pszErrorMsg="[IBM][CLI Driver][DB2/6000] SQL0104N  An unexpected
           token "foo bad sql" was found following "select * ".
           Expected tokens may include:  "<space>".  SQLSTATE=42601
111  ", pcbErrorMsg=163 )
112      <--- SQL_SUCCESS   Time elapsed - +5.299000E-003 seconds

113  SQLError( hEnv=0:0, hDbc=0:0, hStmt=1:1, pszSqlState=&2ff6f19c,
           pfNativeError=&2ff6ed00, pszErrorMsg=&2ff6ed9c, cbErrorMsgMax=1024,
           pcbErrorMsg=&2ff6ed0a )
114      ---> Time elapsed - +2.753000E-003 seconds

115  SQLError( )
```

```
116      <--- SQL_NO_DATA_FOUND   Time elapsed - +2.502000E-003 seconds

117  SQLExecDirect( hStmt=1:1, pszSqlStr="select * foo bad sql", cbSqlStr=-3 )
118      ---> Time elapsed - +3.292000E-003 seconds

119  SQLExecDirect( )
120      <--- SQL_ERROR   Time elapsed - +6.012500E-002 seconds

121  SQLFreeHandle( fHandleType=SQL_HANDLE_STMT, hHandle=1:1 )
122      ---> Time elapsed - +2.867000E-003 seconds
123  ( Unretrieved error message="SQL0104N  An unexpected token "foo bad sql"
           was found following "select * ".  Expected tokens may
           include:  "<space>".  SQLSTATE=42601
124  " )

125  SQLFreeHandle( )
126      <--- SQL_SUCCESS   Time elapsed - +4.936600E-002 seconds

127  SQLEndTran( fHandleType=SQL_HANDLE_DBC, hHandle=0:1, fType=SQL_ROLLBACK )
128      ---> Time elapsed - +2.968000E-003 seconds

129  SQLEndTran( )
130      <--- SQL_SUCCESS   Time elapsed - +1.643370E-001 seconds

131  SQLDisconnect( hDbc=0:1 )
132      ---> Time elapsed - +2.559000E-003 seconds

133  SQLDisconnect( )
134      <--- SQL_SUCCESS   Time elapsed - +8.253310E-001 seconds

135  SQLFreeHandle( fHandleType=SQL_HANDLE_DBC, hHandle=0:1 )
136      ---> Time elapsed - +4.247000E-003 seconds

137  SQLFreeHandle( )
138      <--- SQL_SUCCESS   Time elapsed - +4.742000E-003 seconds

139  SQLFreeHandle( fHandleType=SQL_HANDLE_ENV, hHandle=0:1 )
140      ---> Time elapsed - +2.023000E-003 seconds

141  SQLFreeHandle( )
142      <--- SQL_SUCCESS   Time elapsed - +4.420000E-003 seconds
```

Tracing Muli-Threaded or Muli-Process Applications

For the trace to be of any use for multi-threaded or multi-process applications, you will need to use the TRACEPATHNAME keyword. (Otherwise the trace will be garbled if multiple threads or processes are writing to it simultaneously). See “TRACEPATHNAME” on page 205 for more information.

The files are created in the path specified with the name set to the process id of the application and an extension that is a sequence number for each unique thread (eg. 65397.0, 65397.1, 65397.2 etc.).

By having each thread write to its own file, no semaphores are needed to control access to the tracefile, which means tracing dosn't change the behavior of a multi-thread application. (Of course, tracing may effect the timing of a multi-threaded application).

ODBC Driver Manager Tracing

It is useful to understand the difference between the ODBC trace provided by the ODBC Driver Manager and the DB2 CLI/ODBC driver (IBM ODBC Driver Tracing).

The first thing that is noticable is that the output file formats are different. The important distinction is that the ODBC trace will show the calls made by the application to the Driver Manager. The DB2 CLI trace shows the calls received from the ODBC Driver manager.

The ODBC driver manager may map application function calls to either different functions, different arguments or may delay the call.

One or more of the following may apply:

- Applications written using ODBC 2.0 functions that have been replaced in ODBC 3.0, will have the old functions mapped to the new ones by the ODBC Driver Manager.
- Some function arguments may have their values mapped from ODBC 2.0 values to equivalent ODBC 3.0 values.
- The Microsoft cursor library will map calls such `SQLExtendedFetch()` to multiple calls to fetch, and other suporting functions.

For these reasons you may need to enable and compare the output of both traces to get a clear picture of what is happening.

For more information refer to the *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*.

Appendix L. How the DB2 Library Is Structured

The DB2 Universal Database library consists of SmartGuides, online help, books and sample programs in HTML format. This section describes the information that is provided, and how to access it.

To access product information online, you can use the Information Center. You can view task information, DB2 books, troubleshooting information, sample programs, and DB2 information on the Web. See "Accessing Information with the Information Center" on page 878 for details.

Completing Tasks with SmartGuides

SmartGuides help you complete some administration tasks by taking you through each task one step at a time. SmartGuides are available through the Control Center and the Client Configuration Assistant. The following table lists the SmartGuides.

Note: Create Database, Index, and Configure Multisite Update SmartGuide are available for the partitioned database environment.

SmartGuide	Helps You to...	How to Access...
Add Database	Catalog a database on a client workstation.	From the Client Configuration Assistant, click **Add**.
Back up Database	Determine, create, and schedule a backup plan.	From the Control Center, click with the right mouse button on the database you want to back up and select **Backup**->**Database using SmartGuide**.
Configure Multisite Update SmartGuide	Perform a multi-site update, a distributed transaction, or a two-phase commit.	From the Control Center, click with the right mouse button on the **Database** icon and select **Multisite Update**.
Create Database	Create a database, and perform some basic configuration tasks.	From the Control Center, click with the right mouse button on the **Databases** icon and select **Create**->**Database using SmartGuide**.

SmartGuide	Helps You to...	How to Access...
Create Table	Select basic data types, and create a primary key for the table.	From the Control Center, click with the right mouse button on the **Tables** icon and select **Create->Table using SmartGuide**.
Create Table Space	Create a new table space.	From the Control Center, click with the right mouse button on the **Table spaces** icon and select **Create->Table space using SmartGuide**.
Index	Advise which indexes to create and drop for all your queries.	From the Control Center, click with the right mouse button on the **Index** icon and select **Create->Index using SmartGuide**.
Performance Configuration	Tune the performance of a database by updating configuration parameters to match your business requirements.	From the Control Center, click with the right mouse button on the database you want to tune and select **Configure using SmartGuide**.
Restore Database	Recover a database after a failure. It helps you understand which backup to use, and which logs to replay.	From the Control Center, click with the right mouse button on the database you want to restore and select **Restore->Database using SmartGuide**.

Accessing Online Help

Online help is available with all DB2 components. The following table describes the various types of help. You can also access DB2 information through the Information Center. For information see "Accessing Information with the Information Center" on page 878.

Type of Help	Contents	How to Access...
Command Help	Explains the syntax of commands in the command line processor.	From the command line processor in interactive mode, enter: ? *command* where *command* is 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.

Type of Help	Contents	How to Access...
Control Center Help *Client Configuration Assistant Help* *Event Analyzer Help* *Command Center Help*	Explains the tasks you can perform in a window or notebook. The help includes prerequisite information you need to know, and describes how to use the window or notebook controls.	From a window or notebook, click the **Help** push button or press the F1 key.

Message Help	Describes the cause of a message, and any action you should take.	From the command line processor in interactive mode, enter:
		
		`? XXXnnnnn`
		
		where *XXXnnnnn* is a valid message identifier.
		
		For example, `? SQL30081` displays help about the SQL30081 message.
		
		To view message help one screen at a time, enter:
		
		`? XXXnnnnn | more`
		
		To save message help in a file, enter:
		
		`? XXXnnnnn > filename.ext`
		
		where *filename.ext* is the file where you want to save the message help.

SQL Help	Explains the syntax of SQL statements.	From the command line processor in interactive mode, enter: `help statement` where *statement* is an SQL statement. For example, **help** SELECT displays help about the SELECT statement. **Note:** SQL help is not available on UNIX-based platforms.
SQLSTATE Help	Explains SQL states and class codes.	From the command line processor in interactive mode, enter: **?** *sqlstate* or **?** *class-code* where *sqlstate* is a valid five-digit SQL state and *class-code* is 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.

DB2 Information – Hardcopy and Online

The table in this section lists the DB2 books. They are divided into two groups:

Cross-platform books
These books contain the common DB2 information for all platforms.

Platform-specific books
These books are for DB2 on a specific platform. For example, there are separate *Quick Beginnings* books for DB2 on OS/2, on Windows NT, and on the UNIX-based platforms.

Cross-platform sample programs in HTML
These samples are the HTML version of the sample programs that are installed with the SDK. They are for informational purposes and do not replace the actual programs.

Most books are available in HTML and PostScript format, or you can choose to order a hardcopy from IBM. The exceptions are noted in the table.

On OS/2 and Windows platforms, HTML documentation files can be installed under the doc\html subdirectory. Depending on the language of your system, some files may be in that language, and the remainder are in English.

On UNIX platforms, you can install multiple language versions of the HTML documentation files under the doc/%L/html subdirectories. Any documentation that is not available in a national language is shown in English.

You can obtain DB2 books and access information in a variety of different ways:

View See "Viewing Online Information" on page 877.

Search See "Searching Online Information" on page 880.

Print See "Printing the PostScript Books" on page 880.

Order See "Ordering the Printed Books" on page 881.

Name	Description	Form Number File Name for Online Book	HTML Directory
	Cross-Platform Books		

Name	Description	Form Number File Name for Online Book	HTML Directory
Administration Guide	*Administration Guide, Design and Implementation* contains information required to design, implement, and maintain a database. It also describes database access using the Control Center(whether local or in a client/server environment), auditing, database recovery, distributed database support, and high availability. *Administration Guide, Performance* contains information that focuses on the database environment, such as application performance evaluation and tuning. You can order both volumes of the *Administration Guide* in the English language in North America using the form number SBOF-8922.	Volume 1 SC09-2839 db2d1x60 Volume 2 SC09-2840 db2d2x60	db2d0
Administrative API Reference	Describes the DB2 application programming interfaces (APIs) and data structures you can use to manage your databases. Explains how to call APIs from your applications.	SC09-2841 db2b0x60	db2b0
Application Building Guide	Provides environment setup information and step-by-step instructions about how to compile, link, and run DB2 applications on Windows, OS/2, and UNIX-based platforms. This book combines the *Building Applications* books for the OS/2, Windows, and UNIX-based environments.	SC09-2842 db2axx60	db2ax
APPC, CPI-C and SNA Sense Codes	Provides general information about APPC, CPI-C, and SNA sense codes that you may encounter when using DB2 Universal Database products. **Note:** Available in HTML format only.	No form number db2apx60	db2ap

Name	Description	Form Number File Name for Online Book	HTML Directory
Application Development Guide	Explains how to develop applications that access DB2 databases using embedded SQL or JDBC, how to write stored procedures, user-defined types, user-defined functions, and how to use triggers. It also discusses programming techniques and performance considerations. This book was formerly known as the *Embedded SQL Programming Guide*.	SC09-2845 db2a0x60	db2a0
CLI Guide and Reference	Explains how to develop applications that access DB2 databases using the DB2 Call Level Interface, a callable SQL interface that is compatible with the Microsoft ODBC specification.	SC09-2843 db2l0x60	db2l0
Command Reference	Explains how to use the command line processor, and describes the DB2 commands you can use to manage your database.	SC09-2844 db2n0x60	db2n0
Data Movement Utilities Guide and Reference	Explains how to use the Load, Import, Export, Autoloader, and Data Propogation utilities to work with the data in the database.	SC09-2858 db2dmx60	db2dm
DB2 Connect Personal Edition Quick Beginnings	Provides planning, installing, and configuring information for DB2 Connect Personal Edition.	GC09-2830 db2c1x60	db2c1
DB2 Connect User's Guide	Provides concepts, programming and general usage information about the DB2 Connect products.	SC09-2838 db2c0x60	db2c0
Connectivity Supplement	Provides setup and reference information on how to use DB2 for AS/400, DB2 for OS/390, DB2 for MVS, or DB2 for VM as DRDA application requesters with DB2 Universal Database servers, and on how to use DRDA application servers with DB2 Connect application requesters. **Note:** Available in HTML and PostScript formats only.	No form number db2h1x60	db2h1
Glossary	Provides a comprehensive list of all DB2 terms and definitions. **Note:** Available in HTML format only.	No form number db2t0x50	db2t0

Name	Description	Form Number File Name for Online Book	HTML Directory
Installation and Configuration Supplement	Guides you through the planning, installation, and set up of platform-specific DB2 clients. This supplement contains information on binding, setting up client and server communications, DB2 GUI tools, DRDA AS, distributed installation, and the configuration of distributed requests and access methods to heterogeneous data sources.	GC09-2857 db2iyx60	db2iy
Message Reference	Lists messages and codes issued by DB2, and describes the actions you should take.	GC09-2846 db2m0x60	db2m0
Replication Guide and Reference	Provides planning, configuration, administration, and usage information for the IBM Replication tools supplied with DB2.	SC26-9642 db2e0x60	db2e0
SQL Getting Started	Introduces SQL concepts, and provides examples for many constructs and tasks.	SC09-2856 db2y0x60	db2y0
SQL Reference, Volume 1 and Volume 2	Describes SQL syntax, semantics, and the rules of the language. Also includes information about release-to-release incompatibilities, product limits, and catalog views. You can order both volumes of the *SQL Reference* in the English language in North America with the form number SBOF-8923.	SBOF-8923 Volume 1 db2s1x60 Volume 2 db2s2x60	db2s0
System Monitor Guide and Reference	Describes how to collect different kinds of information about databases and the database manager. Explains how to use the information to understand database activity, improve performance, and determine the cause of problems.	SC09-2849 db2f0x60	db2f0
Troubleshooting Guide	Helps you determine the source of errors, recover from problems, and use diagnostic tools in consultation with DB2 Customer Service.	S10J-8169	db2p0

Name	Description	Form Number File Name for Online Book	HTML Directory
What's New	Describes the new features, functions, and enhancements in DB2 Universal Database, Version 6.0, including information about Java-based tools.	SC09-2851 db2q0x60	db2q0
	Platform-Specific Books		
Administering Satellites Guide and Reference	Provides planning, configuration, administration, and usage information for satellites.	GC09-2821 db2dsx60	db2ds
DB2 Personal Edition Quick Beginnings	Provides planning, installation, migration, and configuration information for DB2 Universal Database Personal Edition on the OS/2, Windows 95, and Windows NT operating systems.	GC09-2831 db2i1x60	db2i1
DB2 for OS/2 Quick Beginnings	Provides planning, installation, migration, and configuration information for DB2 Universal Database on the OS/2 operating system. Also contains installing and setup information for many supported clients.	GC09-2834 db2i2x60	db2i2
DB2 for UNIX Quick Beginnings	Provides planning, installation, migration, and configuration information for DB2 Universal Database on UNIX-based platforms. Also contains installing and setup information for many supported clients.	GC09-2836 db2ixx60	db2ix
DB2 for Windows NT Quick Beginnings	Provides planning, installation, migration, and configuration information for DB2 Universal Database on the Windows NT operating system. Also contains installing and setup information for many supported clients.	GC09-2835 db2i6x60	db2i6
DB2 Enterprise - Extended Edition for UNIX Quick Beginnings	Provides planning, installation, and configuration information for DB2 Enterprise - Extended Edition for UNIX. Also contains installing and setup information for many supported clients.	GC09-2832 db2v3x60	db2v3

Name	Description	Form Number File Name for Online Book	HTML Directory
DB2 Enterprise - Extended Edition for Windows NT Quick Beginnings	Provides planning, installation, and configuration information for DB2 Enterprise - Extended Edition for Windows NT. Also contains installing and setup information for many supported clients.	GC09-2833 db2v6x60	db2v6
DB2 Connect Enterprise Edition for OS/2 and Windows NT Quick Beginnings	Provides planning, migration, installation, and configuration information for DB2 Connect Enterprise Edition on the OS/2 and Windows NT operating systems. Also contains installation and setup information for many supported clients. This book was formerly part of the *DB2 Connect Enterprise Edition Quick Beginnings.*	GC09-2828 db2c6x60	db2c6
DB2 Connect Enterprise Edition for UNIX Quick Beginnings	Provides planning, migration, installation, configuration, and usage information for DB2 Connect Enterprise Edition in UNIX-based platforms. Also contains installation and setup information for many supported clients. This book was formerly part of the *DB2 Connect Enterprise Edition Quick Beginnings.*	GC09-2829 db2cyx60	db2cy
DB2 Data Links Manager for AIX Quick Beginnings	Provides planning, installation, configuration, and task information for DB2 Data Links Manager for AIX.	GC09-2837 db2z0x60	db2z0
DB2 Data Links Manager for Windows NT Quick Beginnings	Provides planning, installation, configuration, and task information for DB2 Data Links Manager for Windows NT.	GC09-2827 db2z6x60	db2z6
DB2 Query Patroller Administration Guide	Provides administration information on DB2 Query Patrol.	SC09-2859 db2dwx60	db2dw
DB2 Query Patroller Installation Guide	Provides installation information on DB2 Query Patrol.	GC09-2860 db2iwx60	db2iw
DB2 Query Patroller User's Guide	Describes how to use the tools and functions of the DB2 Query Patrol.	SC09-2861 db2wwx60	db2ww

Name	Description	Form Number File Name for Online Book	HTML Directory
	Cross-Platform Sample Programs in HTML		
Sample programs in HTML	Provides the sample programs in HTML format for the programming languages on all platforms supported by DB2 for informational purposes (not all samples are available in all languages). Only available when the SDK is installed. See *Application Building Guide* for more information on the actual programs. **Note:** Available in HTML format only.	No form number	db2hs/c db2hs/cli db2hs/clp db2hs/cpp db2hs/cobol db2hs/cobol_mf db2hs/fortran db2hs/java db2hs/rexx

Notes:

1. The character in the sixth position of the file name indicates the language of a book. For example, the file name `db2d0e60` indicates that the *Administration Guide* is in English. The following letters are used in the file names to indicate the language of a book:

Language	Identifier
Brazilian Portuguese	b
Bulgarian	u
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
Russian	r
Simp. Chinese	c
Slovenian	l
Spanish	z

Swedish	s
Trad. Chinese	t
Turkish	m

2. For late breaking information that could not be included in the DB2 books:
   - On UNIX-based platforms, see the Release.Notes file. This file is located in the DB2DIR/Readme/%L directory, where %L is the locale name and DB2DIR is:
     - /usr/lpp/db2_06_01 on AIX
     - /opt/IBMdb2/V6.1 on HP-UX, Solaris, SCO UnixWare 7, and Silicon Graphics IRIX
     - /usr/IBMdb2/V6.1 on Linux.
   - On other platforms, see the RELEASE.TXT file. This file is located in the directory where the product is installed.
   - Under Windows Start menu

Viewing Online Information

The manuals included with this product are in Hypertext Markup Language (HTML) softcopy format. Softcopy format enables you to search or browse the information, and provides hypertext links to related information. It also makes it easier to share the library across your site.

You can view the online books or sample programs with any browser that conforms to HTML Version 3.2 specifications.

To view online books or sample programs on all platforms other than SCO UnixWare 7:

- If you are running DB2 administration tools, use the Information Center. See "Accessing Information with the Information Center" on page 878 for details.
- Select the Open Page menu item of your Web browser. The page you open contains descriptions of and links to DB2 information:
  - On UNIX-based platforms, open the following page:

    ```
    file:/INSTHOME/sqllib/doc/%L/html/index.htm
    ```

    where *%L* is the locale name.
  - On other platforms, open the following page:

    ```
    sqllib\doc\html\index.htm
    ```

    The path is located on the drive where DB2 is installed.

If you have not installed the Information Center, you can open the page by double-clicking on the **DB2 Online Books** icon. Depending on the system you are using, the icon is in the main product folder or the Windows Start menu.

To view online books or sample programs on the SCO UnixWare 7:

- DB2 Universal Database for SCO UnixWare 7 uses the native SCOhelp utility to search the DB2 information. You can access SCOhelp by the following methods:
  - entering the "scohelp" command on the command line,
  - selecting the `Help` menu in the Control Panel of the CDE desktop or
  - selecting `Help` in the Root menu of the Panorama desktop

  For more information on SCOhelp, refer to the *Installation and Configuration Supplement*.

Accessing Information with the Information Center

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

Depending on your system, you can access the Information Center from the:

- Main product folder
- Toolbar in the Control Center
- Windows Start menu
- Help menu of the Control Center

The Information Center provides the following kinds of information. Click the appropriate tab to look at the information:

Tasks	Lists tasks you can perform using DB2.
Reference	Lists DB2 reference information, such as keywords, commands, and APIs.
Books	Lists DB2 books.
Troubleshooting	Lists categories of error messages and their recovery actions.
Sample Programs	Lists sample programs that come with the DB2 Software Developer's Kit. If the Software Developer's Kit is not installed, this tab is not displayed.
Web	Lists DB2 information on the World Wide

Web. To access this information, you must have a connection to the Web from your system.

When you select an item in one of the lists, the Information Center launches a viewer to display the information. The viewer might be the system help viewer, an editor, or a Web browser, depending on the kind of information you select.

The Information Center provides some search capabilities, so you can look for specific topics, and filter capabilities to limit the scope of your searches.

For a full text search, click the Search button of the Information Center follow the *Search DB2 Books* link in each HTML file.

The HTML search server is usually started automatically. If a search in the HTML information does not work, you may have to start the search server by double-clicking its icon on the Windows or OS/2 desktop.

Refer to the release notes if you experience any other problems when searching the HTML information.

Note: Search function is not available in the Linux and Silicon Graphics environments.

Setting Up a Document Server

By default, the DB2 information is installed on your local system. This means that each person who needs access to the DB2 information must install the same files. To have the DB2 information stored in a single location, use the following instructions:

1. Copy all files and subdirectories from \sqllib\doc\html on your local system to a Web server. Each book has its own subdirectory containing all the necessary HTML and GIF files that make up the book. Ensure that the directory structure remains the same.
2. Configure the Web server to look for the files in the new location. For information, see the NetQuestion Appendix in *Installation and Configuration Supplement*.
3. If you are using the Java version of the Information Center, you can specify a base URL for all HTML files. You should use the URL for the list of books.
4. Once you are able to view the book files, you should bookmark commonly viewed topics. Among those, you will probably want to bookmark the following pages:

- List of books
- Tables of contents of frequently used books
- Frequently referenced articles, such as the *ALTER TABLE* topic
- The Search form

For information about setting up a search, see the NetQuestion Appendix in *Installation and Configuration Supplement* book.

Searching Online Information

To search for information in the HTML books, you can do the following:

- Click on **Search the DB2 Books** at the bottom of any page in the HTML books. Use the search form to find a specific topic. This function is not available in the Linux or Silicon Graphics IRIX environments.
- Click on **Index** at the bottom of any page in an HTML book. Use the index to find a specific topic in the book.
- Display the table of contents or index of the HTML book, and then use the find function of the Web browser to find a specific topic in the book.
- Use the bookmark function of the Web browser to quickly return to a specific topic.
- Use the search function of the Information Center to find specific topics. See "Accessing Information with the Information Center" on page 878 for details.

Printing the PostScript Books

If you prefer to have printed copies of the manuals, you can decompress and print PostScript versions. For the file name of each book in the library, see the table in "DB2 Information – Hardcopy and Online" on page 870. Specify the full path name for the file you intend to print.

On OS/2 and Windows platforms:

1. Copy the compressed PostScript files to a hard drive on your system. The files have a file extension of .exe and are located in the x:\doc*language*\books\ps directory, where x: is the letter representing the CD-ROM drive and *language* is the two-character country code that represents your language (for example, EN for English).
2. Decompress the file that corresponds to the book that you want. Each compressed book is a self-extracting executable file. To decompress the

book, simply run it as you would run any other executable program. The result from this step is a printable PostScript file with a file extension of .ps.

3. Ensure that your default printer is a PostScript printer capable of printing Level 1 (or equivalent) files.
4. Enter the following command from a command line:

   ```
   print filename.ps
   ```

On UNIX-based platforms:

1. Mount the CD-ROM. Refer to your *Quick Beginnings* manual for the procedures to mount the CD-ROM.
2. Change to /cdrom/doc/%L/ps directory on the CD-ROM, where */cdrom* is the mount point of the CD-ROM and *%L* is the name of the desired locale. The manuals will be installed in the previously-mentioned directory with file names ending with .ps.Z.
3. Decompress and print the manual you require using the following command:
   - For AIX:

     ```
     zcat filename | qprt -P PSPrinter_queue
     ```

   - For HP-UX, Solaris, or SCO UnixWare 7:

     ```
     zcat filename | lp -d PSPrinter_queue
     ```

   - For Linux:

     ```
     zcat filename | lpr -P PSPrinter_queue
     ```

   - For Silicon Graphics IRIX:

     ```
     zcat < filename | lp -d PSPrinter_queue
     ```

   where *filename* is the full path name and extension of the compressed PostScript file and *PSprinter_queue* is the name of the PostScript printer queue.

   For example, to print the English version of *DB2 for UNIX Quick Beginnings* on AIX, you can use the following command:

   ```
   zcat /cdrom/doc/en/ps/db2ixe60.ps.Z || qprt -P ps1
   ```

Ordering the Printed Books

You can order the printed DB2 manuals either as a set or individually. There are three sets of books available. The form number for the entire set of DB2 books is SBOF-8926-00. The form number for the books listed under the heading "Cross-Platform Books" is SBOF-8924-00.

Note: These form numbers only apply if you are ordering books that are printed in the English language in North America.

You can also order books individually by the form number listed in “DB2 Information – Hardcopy and Online” on page 870. To order printed versions, contact your IBM authorized dealer or marketing representative, or phone `1-800-879-2755` in the United States or `1-800-IBM-4YOU` in Canada.

Appendix M. Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter 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 the

IBM Director of Licensing
IBM Corporation, North Castle Drive
Armonk, NY 10504-1785
U.S.A.

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
1150 Eglinton Ave. East
North York, Ontario
M3C 1H7
CANADA

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

This publication 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.

This book incorporates text which is copyright The X/Open Company Limited. The text was taken by permission from:

X/Open CAE Specification, March 1995,
Data Management: SQL Call Level Interface (CLI)
(ISBN: 1-85912-081-4, C451).

X/Open Preliminary Specification, March 1995,
Data Management: Structured Query Language (SQL), Version 2
(ISBN: 1-85912-093-8, P446).

Trademarks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries:

ACF/VTAM
ADSTAR
AISPO
AIX
AIXwindows
AnyNet
APPN
AS/400
CICS
C Set++
C/370
DATABASE 2
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
DB2 Connect
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
Extended Services
FFST
First Failure Support Technology
IBM
IMS
LAN Distance
MVS/ESA
MVS/XA
OS/400
OS/390
OS/2
PowerPC
QMF
RACF
RISC System/6000
SP
SQL/DS
SQL/400
S/370
System/370
System/390
SystemView
VisualAge
VM/ESA
VSE/ESA
VTAM
WIN-OS/2

Trademarks of Other Companies

The following terms are trademarks or registered trademarks of the companies listed:

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

HP-UX is a trademark of Hewlett-Packard.

Java, HotJava, Solaris, Solstice, and Sun are trademarks of Sun Microsystems, Inc.

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

PC Direct is a trademark of Ziff Communications Company in the United States, other countries, or both and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium, and ProShare are trademarks of Intel Corporation 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, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Bibliography

- *IBM SQL Reference Version 2*, SC26-8416
- *ODBC 2.0 Programmer's Reference and SDK Guide* ISBN 1-55615-658-8
- *ODBC 3.0 Software Development Kit and Programmer's Reference* ISBN 1-57231-516-4
- X/Open CAE Specification, March 1995, *Data Management: SQL Call Level Interface (CLI)* (ISBN: 1-85912-081-4, C451).
- X/Open Preliminary Specification, March 1995, *Data Management: Structured Query Language (SQL), Version 2* (ISBN: 1-85912-093-8, P446).
- *IBM DB2 SQL Reference* SC09-2847
- *IBM DATABASE 2 for MVS/ESA SQL Reference* SC26-3270-00
- *SQL/DS SQL Reference for IBM VM Systems and VSE* SH09-8087-03
- *DB2 for OS/400 SQL Reference Version 3* SC41-9608-00

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Contacting IBM

This section lists ways you can get more information from IBM.

If you have a technical problem, please take the time to review and carry out the actions suggested by the *Troubleshooting Guide* before contacting DB2 Customer Support. Depending on the nature of your problem or concern, this guide will suggest information you can gather to help us to serve you better.

For information or to order any of the DB2 Universal Database products contact an IBM representative at a local branch office or contact any authorized IBM software remarketer.

Telephone

If you live in the U.S.A., call one of the following numbers:

- 1-800-237-5511 to learn about available service options.
- 1-800-IBM-CALL (1-800-426-2255) or 1-800-3IBM-OS2 (1-800-342-6672) to order products or get general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, see Appendix A of the IBM Software Support Handbook. You can access this document by accessing the following page:

`http://www.ibm.com/support/`

then performing a search using the keyword “handbook”.

Note that in some countries, IBM-authorized dealers should contact their dealer support structure instead of the IBM Support Center.

World Wide Web

http://www.software.ibm.com/data/
http://www.software.ibm.com/data/db2/library/

The DB2 World Wide Web pages provide current DB2 information about news, product descriptions, education schedules, and more. The DB2 Product and Service Technical Library provides access to frequently asked questions, fixes, books, and up-to-date DB2 technical information. (Note that this information may be in English only.)

Anonymous FTP Sites

ftp.software.ibm.com

Log on as anonymous. In the directory /ps/products/db2, you can find demos, fixes, information, and tools concerning DB2 and many related products.

Internet Newsgroups

comp.databases.ibm-db2, bit.listserv.db2-l

These newsgroups are available for users to discuss their experiences with DB2 products.

CompuServe

GO IBMDB2 to access the IBM DB2 Family forums

All DB2 products are supported through these forums.

To find out about the IBM Professional Certification Program for DB2 Universal Database, go to http://www.software.ibm.com/data/db2/db2tech/db2cert.html

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC09-2843-00