DB2® Universal Database for z/OS

IBM

Version 8

ODBC Guide and Reference

SC18-7423-00

DB2® Universal Database for z/OS

Version 8

ODBC Guide and Reference

SC18-7423-00

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 559.

First Edition (March 2004)

This edition applies to Version 8 of IBM DB2 Universal Database for z/OS (DB2 UDB for z/OS), product number 5625-DB2, and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Specific changes are indicated by a vertical bar to the left of a change. A vertical bar to the left of a figure caption indicates that the figure has changed. Editorial changes that have no technical significance are not noted.

Contents

About this book

This book provides the information necessary to write applications using DB2® ODBC to access IBM DB2 servers, as well as any database that supports DRDA® level 1 or DRDA level 2 protocols. This book should also be used as a supplement when writing portable ODBC applications that can be executed in a native DB2 UDB for z/OS environment using DB2 ODBC.

Important

In this version of DB2 UDB for z/OS, the DB2 Utilities Suite is available as an optional product. You must separately order and purchase a license to such utilities, and discussion of those utility functions in this publication is not intended to otherwise imply that you have a license to them. See Part 1 of *DB2 Utility Guide and Reference* for packaging details.

Who should use this book

This book is intended for the following audiences:

- DB2 application programmers with a knowledge of SQL and the C programming language.
- ODBC application programmers with a knowledge of SQL and the C programming language.

Terminology and citations

In this information, DB2 Universal Database™ for z/OS™ is referred to as "DB2 UDB for z/OS." In cases where the context makes the meaning clear, DB2 UDB for z/OS is referred to as "DB2." When this information refers to titles of books in this library, a short title is used. (For example, "See *DB2 SQL Reference*" is a citation to *IBM® DB2 Universal Database for z/OS SQL Reference*.)

When referring to a DB2 product other than DB2 UDB for z/OS, this information uses the product's full name to avoid ambiguity.

The following terms are used as indicated:

DB2 Represents either the DB2 licensed program or a particular DB2 subsystem.

DB2 PM
Refers to the DB2 Performance Monitor tool, which can be used on its own or as part of the DB2 Performance Expert for z/OS product.

C, C++, and C language
Represent the C or C++ programming language.

CICS® Represents CICS Transaction Server for z/OS or CICS Transaction Server for OS/390®.

IMS™ Represents the IMS Database Manager or IMS Transaction Manager.

MVS™ Represents the MVS element of the z/OS operating system, which is equivalent to the Base Control Program (BCP) component of the z/OS operating system.

RACF®
: Represents the functions that are provided by the RACF component of the z/OS Security Server.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products. The major accessibility features in z/OS products, including DB2 UDB for z/OS, enable users to:

- Use assistive technologies such as screen reader and screen magnifier software
- Operate specific or equivalent features by using only a keyboard
- Customize display attributes such as color, contrast, and font size

Assistive technology products, such as screen readers, function with the DB2 UDB for z/OS user interfaces. Consult the documentation for the assistive technology products for specific information when you use assistive technology to access these interfaces.

Online documentation for Version 8 of DB2 UDB for z/OS is available in the DB2 Information Center, which is an accessible format when used with assistive technologies such as screen reader or screen magnifier software. The DB2 Information Center for z/OS solutions is available at the following Web site: http://publib.boulder.ibm.com/infocenter/db2zhelp.

How to send your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 UDB for z/OS documentation. You can use the following methods to provide comments:

- Send your comments by e-mail to db2pubs@vnet.ibm.com and include the name of the product, the version number of the product, and the number of the book. If you are commenting on specific text, please list the location of the text (for example, a chapter and section title, page number, or a help topic title).
- You can also send comments from the Web. Visit the library Web site at:

  www.ibm.com/software/db2zos/library.html

  This Web site has a feedback page that you can use to send comments.
- Print and fill out the reader comment form located at the back of this book. You can give the completed form to your local IBM branch office or IBM representative, or you can send it to the address printed on the reader comment form.

Summary of changes to this book

The major changes to this edition of the book are:

- Support for the Unicode UTF-8 encoding format and support for ASCII. “Handling application encoding schemes” on page 443 describes Unicode and ASCII support.
- A new keyword, CURRENTAPPENSCH, that specifies the application encoding scheme in the initialization file. See “Initialization keywords” on page 51.
- Additional SQLGetInfo() attributes to query the CCSID settings of the DB2 subsystem in each encoding scheme. See “SQLGetInfo() - Get general information” on page 234.

Chapter 1. Introduction to DB2 ODBC

DB2 Open Database Connectivity (ODBC) is the IBM callable SQL interface by the DB2 family of products. It is a C and C++ language 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 a precompiler.

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

DB2 ODBC background information

To understand DB2 ODBC 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 product vendor's programming interface. Most of the X/Open Call Level Interface specification was accepted as part of the ISO Call Level Interface Draft International Standard (ISO CLI DIS).

Microsoft® developed a callable SQL interface called Open Database Connectivity (ODBC) for Microsoft operating systems based on a preliminary draft of X/Open CLI. The Call Level Interface specifications in ISO, X/Open, ODBC, and DB2 ODBC continue to evolve in a cooperative manner to provide functions with additional capabilities.

The ODBC specification also includes an operating environment where data source specific ODBC drivers are dynamically loaded at run time by a driver manager based on the data source 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.

The ODBC driver manager only knows about the ODBC-specific functions, that is, those functions supported by the DBMS for which no API is specified. Therefore, DBMS-specific functions cannot be directly accessed in an ODBC environment. However, DBMS-specific dynamic SQL statements are indirectly supported using a mechanism called the vendor escape clause. See "Using vendor escape clauses" on page 465 for detailed information.

ODBC is not limited to Microsoft operating systems. Other implementations are available, such as DB2 ODBC, or are emerging on various platforms.

Differences between DB2 ODBC and ODBC version 3.0

DB2 ODBC is derived from the ISO Call Level Interface Draft International Standard
| (ISO CLI DIS) and ODBC Version 3.0.

If you port existing ODBC applications to DB2 UDB for z/OS or write a new application according to the ODBC specifications, you must comply with the specifications defined in this publication. However, before you write to any API, validate that the API is supported by DB2 ODBC and that the syntax and semantics are identical. For any differences, you must code to the APIs documented in this publication.

On the DB2 UDB for z/OS platform, no ODBC driver manager exists. Consequently, DB2 ODBC support is implemented as a CLI/ODBC driver/driver manager that is loaded at run time into the application address space. See "DB2 ODBC and ODBC drivers" on page 489 for details about the difference between ODBC drivers with and without a driver manager. For details about the DB2 ODBC run-time environment, see "The DB2 ODBC run-time environment" on page 39.

The DB2 UDB for Linux, UNIX and Windows support for CLI executes on Windows and AIX® as an ODBC driver, loaded by the Windows driver manager (Windows environment) or the Visigenic driver manager (UNIX® platforms). In this context, DB2 ODBC support is limited to the ODBC specifications. Alternatively, an application can directly invoke the CLI application programming interfaces (APIs) including those not supported by ODBC. In this context, the set of APIs supported by DB2 UDB is referred to as the ″Call Level Interface.″ See *DB2 Universal Database Call Level Interface Guide and Reference, Volumes 1 and 2*.

The use of DB2 ODBC in this publication refers to DB2 UDB for z/OS support of DB2 ODBC unless otherwise noted.

General information about DB2 UDB for z/OS is available from the DB2 UDB for z/OS web page: www.ibm.com/software/data/db2/zos/.

ODBC features supported

DB2 ODBC support should be viewed as consisting of most of ODBC Version 3.0 as well as IBM extensions. Where differences exist, applications should be written to the specifications defined in this publication.

DB2 ODBC supports the following ODBC functionality:

- ODBC core conformance with the following exceptions:
  - Manipulating fields of descriptors is not supported. DB2 ODBC does not support SQLCopyDesc(), SQLGetDescField(), SQLGetDescRec(), SQLSetDescField(), or SQLSetDescRec().
  - Driver management is not supported. The ODBC driver manager and support for SQLDrivers() is not applicable in the DB2 UDB for z/OS ODBC environment.
- ODBC level 1 conformance with the following exceptions:
  - Asynchronous execution of ODBC functions for individual connections is not supported.
  - SQLSetPos() is not supported. SQLSetPos() positions a cursor, updates rows, and deletes rows in a row set.
  - SQLFetchScroll(), which provides ODBC scrollable cursors, is not supported; access to result sets is limited to forward-only.
  - Connecting interactively to data sources is not supported. DB2 ODBC does not support SQLBrowseConnect() and supports SQLDriverConnect() with SQL_DRIVER_NOPROMPT only.
- ODBC level 2 conformance with the following exceptions:

  - Asynchronous execution of ODBC functions for individual statements is not supported.
  - Bookmarks are not supported. DB2 ODBC does not support SQLFetchScroll() with SQL_FETCH_BOOKMARK; SQLBulkOperations() with SQL_UPDATE_BY_BOOKMARK, SQL_DELETE_BY_BOOKMARK, or SQL_FETCH_BY_BOOKMARK; or retrieving bookmarks on column 0 with SQLDescribeColumn() and SQLColAttribute().
  - SQLBulkOperations() with SQL_ADD, and SQLSetPos() with SQL_DELETE and SQL_UPDATE are not supported.
  - The SQL_ATTR_LOGIN_TIMEOUT connection attribute, which times out login requests, and the SQL_ATTR_QUERY_TIMEOUT statement attribute, which times out SQL queries, are not supported.
- Some X/Open CLI functions
- Some DB2 specific functions

For a complete list of supported functions, see “Function overview” on page 64.

The following DB2 features are available to both ODBC and DB2 ODBC applications:
- The double-byte (graphic) data types (see “GRAPHIC keyword” on page 56)
- Stored procedures (see 429)
- Distributed unit of work (DUW) as defined by DRDA, two-phase commit (see 399)
- Distinct types (see 426)
- User-defined functions (see 426)
- Unicode and ASCII support (see 443)

DB2 ODBC contains extensions to access DB2 features that can not be accessed by ODBC applications:
- SQLCA access for detailed DB2 specific diagnostic information (see 265)
- Control over nul-termination of output strings (see 34)
- Support for large objects (LOBs) and LOB locators (see 423)

For more information about the relationship between DB2 ODBC and ODBC, see Appendix A, “DB2 ODBC and ODBC,” on page 489.

Differences between DB2 ODBC 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 data source, and executed. In contrast, a DB2 ODBC 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 ODBC enables you to write portable applications that are independent of any particular database product. Because you do not precompile ODBC applications, the DB2 ODBC driver imposes a fixed set of precompiler options on statements that you execute through ODBC. These options are intended for general ODBC applications.

This independence means DB2 ODBC applications do not have to be recompiled or rebound to access different DB2 or DRDA data sources, but rather just connect to the appropriate data source at run time.

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

- DB2 ODBC does not require the explicit declaration of cursors. They are generated by DB2 ODBC 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 ODBC. Instead, the execution of a SELECT automatically causes a cursor to be opened.
- Unlike embedded SQL, DB2 ODBC allows the use of parameter markers on the equivalent of the EXECUTE IMMEDIATE statement (the SQLExecDirect() function).
- A COMMIT or ROLLBACK in DB2 ODBC is issued using the SQLEndTran() function call rather than by passing it as an SQL statement.
- DB2 ODBC 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.
- DB2 ODBC 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, differences do exist (some ODBC SQLSTATEs and X/Open defined SQLSTATEs also differ). Refer to Table 233 on page 497 for a cross reference of all DB2 ODBC SQLSTATEs.

Despite these differences, embedded SQL and DB2 ODBC share the following concept in common: DB2 ODBC can execute any SQL statement that can be prepared dynamically in embedded SQL.

Table 1 lists each DB2 UDB for z/OS SQL statement and indicates whether you can execute that statement with DB2 ODBC.

Each DBMS might have additional statements that can be dynamically prepared, in which case DB2 ODBC passes them to the DBMS.

Exception: COMMIT and ROLLBACK can be dynamically prepared by some DBMSs but are not passed. The SQLEndTran() function should be used instead to specify either COMMIT or ROLLBACK.

Table 1. SQL statements

SQL statement	Dynamic[1]	DB2 ODBC[2]
ALTER TABLE	Yes	Yes
ALTER DATABASE	Yes	Yes
ALTER INDEX	Yes	Yes
ALTER STOGROUP	Yes	Yes
ALTER TABLESPACE	Yes	Yes
BEGIN DECLARE SECTION[3]	No	No
CALL	No	Yes[4]

Table 1. SQL statements (continued)

SQL statement	Dynamic[1]	DB2 ODBC[2]
CLOSE	No	SQLFreeHandle()
COMMENT ON	Yes	Yes
COMMIT	Yes	SQLEndTran()
CONNECT (type 1)	No	SQLConnect(), SQLDriverConnect()
CONNECT (type 2)	No	SQLConnect(), SQLDriverConnect()
CREATE { ALIAS, DATABASE, INDEX, STOGROUP, SYNONYM, TABLE, TABLESPACE, VIEW, DISTINCT TYPE }	Yes	Yes
DECLARE CURSOR[3]	No	SQLAllocHandle()
DECLARE STATEMENT	No	No
DECLARE TABLE	No	No
| DECLARE VARIABLE	No	No
DELETE	Yes	Yes
DESCRIBE	No	SQLDescribeCol(), SQLColAttribute()
DROP	Yes	Yes
END DECLARE SECTION[3]	No	No
EXECUTE	No	SQLExecute()
EXECUTE IMMEDIATE	No	SQLExecDirect()
EXPLAIN	Yes	Yes
FETCH	No	SQLFetch(), SQLExtendedFetch()
FREE LOCATOR[4]	No	Yes
| GET DIAGNOSTICS	No	No
GRANT	Yes	Yes
HOLD LOCATOR[4]	No	Yes
INCLUDE[3]	No	No
INSERT	Yes	Yes
LABEL ON	Yes	Yes
LOCK TABLE	Yes	Yes
OPEN	No	SQLExecute(), SQLExecDirect()
PREPARE	No	SQLPrepare()
RELEASE	No	No
REVOKE	Yes	Yes
ROLLBACK	Yes	SQLEndTran()
select-statement	Yes	Yes
SELECT INTO	No	No
SET CONNECTION	No	SQLSetConnection()
SET host_variable	No	No
| SET CURRENT APPLICATION ENCODING SCHEME	No	No
SET CURRENT DEGREE	Yes	Yes

Table 1. SQL statements (continued)

SQL statement	Dynamic[1]	DB2 ODBC[2]
SET CURRENT PACKAGESET	No	No
SET CURRENT PATH	Yes	Yes
SET CURRENT SQLID	Yes	Yes
UPDATE	Yes	Yes
WHENEVER[3]	No	No

Notes:

1. All statements in this list can be coded as static SQL, but only those marked with X can be coded as dynamic SQL.
2. An X indicates that this statement can be executed using either SQLExecDirect(), or SQLPrepare() and SQLExecute(). Equivalent DB2 ODBC functions are listed.
3. This statement is not executable.
4. Although this statement is not dynamic, DB2 ODBC allows the statement to be specified when calling either SQLExecDirect() or SQLPrepare() and SQLExecute().

Advantages of using DB2 ODBC

DB2 ODBC provides a number of key features that offer advantages in contrast to embedded SQL. DB2 ODBC:

- Ideally suits the client-server environment in which the target data source is unknown when the application is built. It provides a consistent interface for executing SQL statements, regardless of which database server the application connects to.
- Lets you write portable applications that are independent of any particular database product. DB2 ODBC applications do not have to be recompiled or rebound to access different DB2 or DRDA data sources. Instead they connect to the appropriate data source at run time.
- Reduces the amount of management required for an application while in general use. Individual DB2 ODBC applications do not need to be bound to each data source. Bind files provided with DB2 ODBC need to be bound only once for all DB2 ODBC applications.
- Lets applications connect to multiple data sources from the same application.
- Allocates and controls data structures, and provides a handle for the application to refer to them. Applications do not have to control complex global data areas such as the SQLDA and SQLCA.
- Provides enhanced parameter input and fetching capability. You can specify arrays of data on input to retrieve multiple rows of a result set directly into an array. You can execute statements that generate multiple result sets.
- Lets you retrieve multiple rows and result sets generated from a call to a stored procedure.
- Provides a consistent interface to query catalog information that is contained in various DBMS catalog tables. The result sets that are returned are consistent across DBMSs. Application programmers can avoid writing version-specific and server-specific catalog queries.
- Provides extended data conversion which requires less application code when converting information between various SQL and C data types.
- Aligns with the emerging ISO CLI standard in addition to using the accepted industry specifications of ODBC and X/Open CLI.

- Allows application developers to apply their knowledge of industry standards directly to DB2 ODBC. The interface is intuitive for programmers who are familiar with function libraries but know little about product specific methods of embedding SQL statements into a host language.

Choosing between SQL and DB2 ODBC

DB2 ODBC is ideally suited for query-based applications that require portability. Use the following guidelines to help you decide which interface meets your needs.

Static and dynamic SQL

Only embedded SQL applications can use static SQL. Both static and dynamic SQL have advantages. Consider these factors:

- Performance

  Dynamic SQL is prepared at run time. Static SQL is prepared at bind time. The preparation step for dynamic SQL requires more processing and might incur additional network traffic.

  However, static SQL does not always perform better than dynamic SQL. Dynamic SQL can make use of changes to the data source, such as new indexes, and can use current catalog statistics to choose the optimal access plan.
- Encapsulation and security

  In static SQL, authorization to objects is associated with a package and validated at package bind time. Database administrators can grant execute authority on a particular package to a set of users rather than grant explicit access to each database object.

  In dynamic SQL, authorization is validated at run time on a per statement basis; therefore, users must be granted explicit access to each database object.

Use both interfaces

An application can take advantage of both static and dynamic interfaces. An application programmer can create a stored procedure that contains static SQL. The stored procedure is called from within a DB2 ODBC application and executed on the server. After the stored procedure is created, any DB2 ODBC or ODBC application can call it.

Write a mixed application

You can write a mixed application that uses both DB2 ODBC and embedded SQL. In this scenario, DB2 ODBC provides the base application, and you write key modules using static SQL for performance or security. Choose this option only if stored procedures do not meet your applications requirements.

DB2 ODBC does not support embedded SQL statements in a multiple context environment. See “DB2 ODBC support of multiple contexts” on page 435 and “Mixing embedded SQL with DB2 ODBC” on page 463 for more information.

Additional DB2 ODBC resources

Application developers should refer to *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference* as a supplement to this publication.

When writing DB2 ODBC applications, you also might 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 UDB for z/OS versions, see *DB2 SQL Reference* and *DB2 Application Programming and SQL Guide*. If you are writing applications that access other DB2 server products, see *IBM SQL Reference* for information that is common to all products, including any differences.

Chapter 2. Writing a DB2 ODBC application

This chapter introduces a conceptual view of a typical DB2 ODBC application.

You can consider a DB2 ODBC application as a set of tasks. Some of these tasks consist of discrete steps, while others might apply throughout the application. One or more DB2 ODBC functions carry out each of these core tasks.

This section describes the basic tasks that apply to all DB2 ODBC applications. Chapter 5, “Using advanced features,” on page 397 describes more advanced tasks, such as using array insert.

Every DB2 ODBC application performs three core tasks: initialization, transaction processing, and termination. Figure 1 illustrates an ODBC application in terms of these tasks.

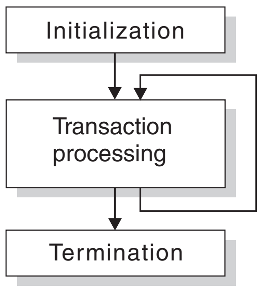


Figure 1. Conceptual view of a DB2 ODBC application

Initialization
: This task allocates and initializes some resources in preparation for the transaction processing task. See “Initialization and termination” on page 10 for details.

Transaction processing
: This task provides functionality to the application. It passes SQL statements to DB2 ODBC that query and modify data. See “Transaction processing” on page 15 for details.

Termination
: This task frees allocated resources. The resources generally consist of data areas identified by unique handles. See “Initialization and termination” on page 10 for details.

In addition to the three tasks listed above, general tasks, such as handling diagnostic messages, occur throughout an application.

Examples in this chapter illustrate the use of functions in DB2 ODBC applications. See Chapter 4, “Functions,” on page 63 for a complete description and usage information for each function that appears in these examples.

Initialization and termination

Figure 2 shows the function call sequences for both the initialization and termination tasks. Figure 4 on page 15 details the transaction processing task in the middle of the diagram, which is not a topic in this section.

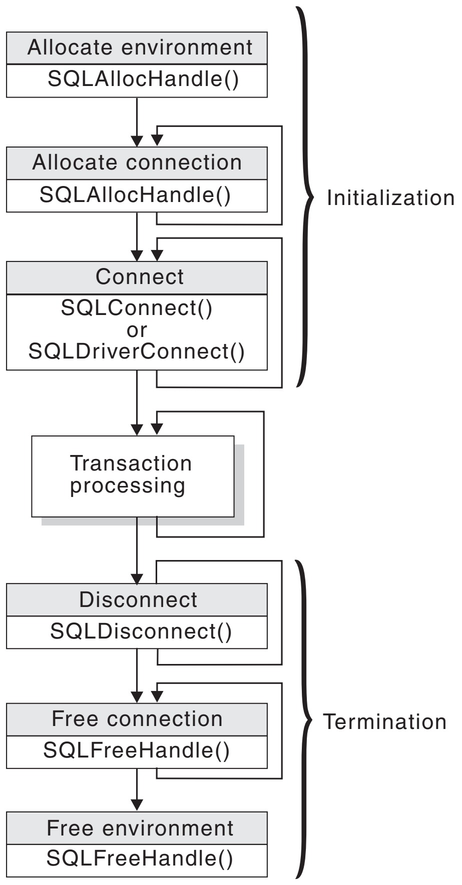


Figure 2. Conceptual view of initialization and termination tasks

In the initialization task, an application allocates handles and connects to data sources. In the termination task, an application frees handles and disconnects from data sources. Use handles and the ODBC connection model to initialize and terminate an application.

Handles

A handle is a variable that refers to a data object controlled by DB2 ODBC. Using handles relieves the application from having to allocate and manage global variables or data structures, such as the SQLDA or SQLCA, that the IBM embedded SQL interfaces use.

DB2 ODBC defines the three following 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. This handle is allocated by calling SQLAllocHandle() (with *HandleType* set to SQL_HANDLE_ENV), and freed by calling

SQLFreeHandle() (with *HandleType* set to SQL_HANDLE_ENV). 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. This includes connection attributes, general status information, transaction status, and diagnostic information. Each connection handle is allocated by calling SQLAllocHandle() (with *HandleType* set to SQL_HANDLE_DBC) and freed by calling SQLFreeHandle() (with *HandleType* set to SQL_HANDLE_DBC).

An application can be connected to several database servers at the same time. An application requires a connection handle for each concurrent connection to a database server. For information about multiple connections, see "Connecting to one or more data sources" on page 12.

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

Statement handles
Statement handles are discussed in the next section, "Transaction processing" on page 15.

The initialization task consists of the allocation and initialization of environment and connection handles. The termination task later frees these handles. An application then passes the appropriate handle when it calls other DB2 ODBC functions.

ODBC connection model

The ODBC specifications support any number of concurrent connections, each of which is an independent transaction. That is, an application can issue SQLConnect() to X, perform some work, issue SQLConnect() to Y, perform some work, and then commit the work at X. ODBC supports multiple concurrent and independent transactions, one per connection.

DB2 ODBC restrictions on the ODBC connection model

If the initialization file does not specify MULTICONTEXT=1, DB2 ODBC does not fully support the ODBC connection model. In this case, to obtain simulated support of the ODBC connection model, an application must specify CONNECTTYPE=1 either through the initialization file or the SQLSetConnectAttr() API. For details, see "Initialization keywords" on page 51 and "Specifying the connection type" on page 12.

An application that uses DB2 ODBC to simulate support of the ODBC model can logically connect to any number of data sources. However, the DB2 ODBC driver maintains only one physical connection. This single connection is to the data source to which the application last successfully connected or issued an SQL statement.

An application that operates with simulated support of the ODBC connection model, regardless of the commit mode, behaves as follows:

- When the application accesses multiple data sources, it allocates a connection handle to each data source. Because this application can make only one physical connection at a time, the DB2 ODBC driver commits the work on the current data source and terminates the current connection before the application connects to a new data source. Therefore, an application that operates with simulated support of the ODBC connection model cannot open cursors concurrently at two data sources (including cursors WITH HOLD).

- When the application does not explicitly commit or rollback work on the current connection before it calls a function on another connection, the DB2 ODBC driver implicitly performs the following actions:
  1. Commits work on the current connection
  2. Disconnects from the current data source
  3. Connects to the new data source
  4. Executes the function

When you enable multiple-context support (MULTICONTEXT=1), DB2 ODBC fully supports the ODBC connection model. See “DB2 ODBC support of multiple contexts” on page 435 for details.

Specifying the connection type

Every IBM RDBMS supports both type 1 and type 2 connection type semantics. In both cases, only one transaction is active at any time.

In SQL, CONNECT (type 1) lets the application connect to only a single database at any time so a single transaction is active on the current connection. This connection type models DRDA remote unit of work processing.

Conversely, CONNECT (type 2), in SQL, lets the application connect concurrently to any number of database servers, all of which participate in a single transaction. This connection type models DRDA distributed unit of work processing.

DB2 ODBC supports both these connection types, but all connections in your application must use only one connection type at a given time. You must free all current connection handles before you change the connection type.

Important: Establish a connection type before you issue SQLConnect().

You can establish the connection type with either of the following methods:
- Specify CONNECTTYPE=1 (for CONNECT (type 1)) or CONNECTTYPE=2 (for CONNECT (type 2)) in the common section of the initialization file. “DB2 ODBC initialization file” on page 49 describes the initialization file.
- Invoke SQLSetConnectAttr() with the *Attribute* argument set to SQL_ATTR_CONNECTTYPE and*ValuePtr* set to SQL_CONCURRENT_TRANS (for CONNECT (type 1)) or SQL_COORDINATED_TRANS (for CONNECT (type 2)).

Connecting to one or more data sources

DB2 ODBC supports connections to remote data sources through DRDA.

If an application is CONNECT (type 1) and specifies MULTICONTEXT=0, DB2 ODBC allows the application to *logically* connect to multiple data sources. However, DB2 ODBC allows the application only one outstanding transaction (a transaction the application has not yet committed or rolled back) on the active connection. If the application is CONNECT (type 2), then the transaction is a distributed unit of work and all data sources participate in the disposition of the transaction (commit or rollback).

To connect concurrently to one or more data sources, call SQLAllocHandle() (with *HandleType* set to SQL_HANDLE_DBC) once for each connection. Use the connection handle that this statement yields in an SQLConnect() call to request a data source connection. Use the same connection handle in an SQLAllocHandle() call (with *HandleType* set to SQL_HANDLE_STMT) to allocate statement handles to use within that connection. An extended connect function, SQLDriverConnect(), allows you to set additional connection attributes.

Unlike the distributed unit of work connections that are described in "Using a distributed unit of work" on page 399, statements that execute on different connections do not coordinate.

Example: Figure 3 illustrates an application that connects, allocates handles, frees handles, and disconnects. This application connects to multiple data sources but does not explicitly set a connection type or specify multiple-context support. The CONNECTTYPE and MULTICONTEXT keywords in the initialization file declare these settings.

```
/* ... */
/********************************************************
**    - Demonstrate basic connection to two data sources.
**    - Error handling mostly ignored for simplicity
**
**  Functions used:
**
**    SQLAllocHandle   SQLDisconnect
**    SQLConnect       SQLFreeHandle
**  Local Functions:
**    DBconnect
**
********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "sqlcli1.h"

int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server);

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

int
main( )
{
    SQLHENV         henv;
    SQLHDBC         hdbc[MAX_CONNECTIONS];
    char *          svr[MAX_CONNECTIONS] =
                    {
                      "KARACHI"   ,
                      "DAMASCUS"
                    }

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

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

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

Figure 3. ODBC application that connects to two data sources. (Part 1 of 2)

```
    /*********   Start processing step  *************************/
    /* Allocate statement handle, execute statement, and so on  */
    /*********   End processing step  ***************************/

    /***********************************************************/
    /* Commit work on connection 1.                            */
    /***********************************************************/

    SQLEndTran(SQL_HANDLE_DBC, hdbc[0], SQL_COMMIT);

    /***********************************************************/
    /* Commit work on connection 2. This has NO effect on the   */
    /* transaction active on connection 1.                     */
    /***********************************************************/

    SQLEndTran(SQL_HANDLE_DBC, hdbc[1], SQL_COMMIT);

    printf("\nDisconnecting .....\n");

    SQLDisconnect(hdbc[0]);   /* disconnect first connection */
       SQLDisconnect(hdbc[1]);     /* disconnect second connection  */

    SQLFreeHandle (SQL_HANDLE_DBC, hdbc[0]);    /* free first connection handle */
    SQLFreeHandle (SQL_HANDLE_DBC, hdbc[1]);    /* free second connection handle */

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

    return (SQL_SUCCESS);
}


/*******************************************************************
**   Server is passed as a parameter. Note that NULL values are    **
**   passed for USERID and PASSWORD.                               **
*******************************************************************/

int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server)
{
    SQLRETURN       rc;
    SQLCHAR         buffer[255];
    SQLSMALLINT     outlen;


    SQLAllocHandle(SQL_HANDLE_DBC, henv, hdbc); /* allocate connection handle */

    rc = SQLConnect(*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf(">--- Error while connecting to database: %s -------\n", server);
        return (SQL_ERROR);
    } else {
        printf(">Connected to %s\n", server);
        return (SQL_SUCCESS);
    }
}
/* ... */
```

Figure 3. ODBC application that connects to two data sources. (Part 2 of 2)

Transaction processing

Figure 4 shows the typical order of function calls in a DB2 ODBC application. It does not show all functions or possible paths.

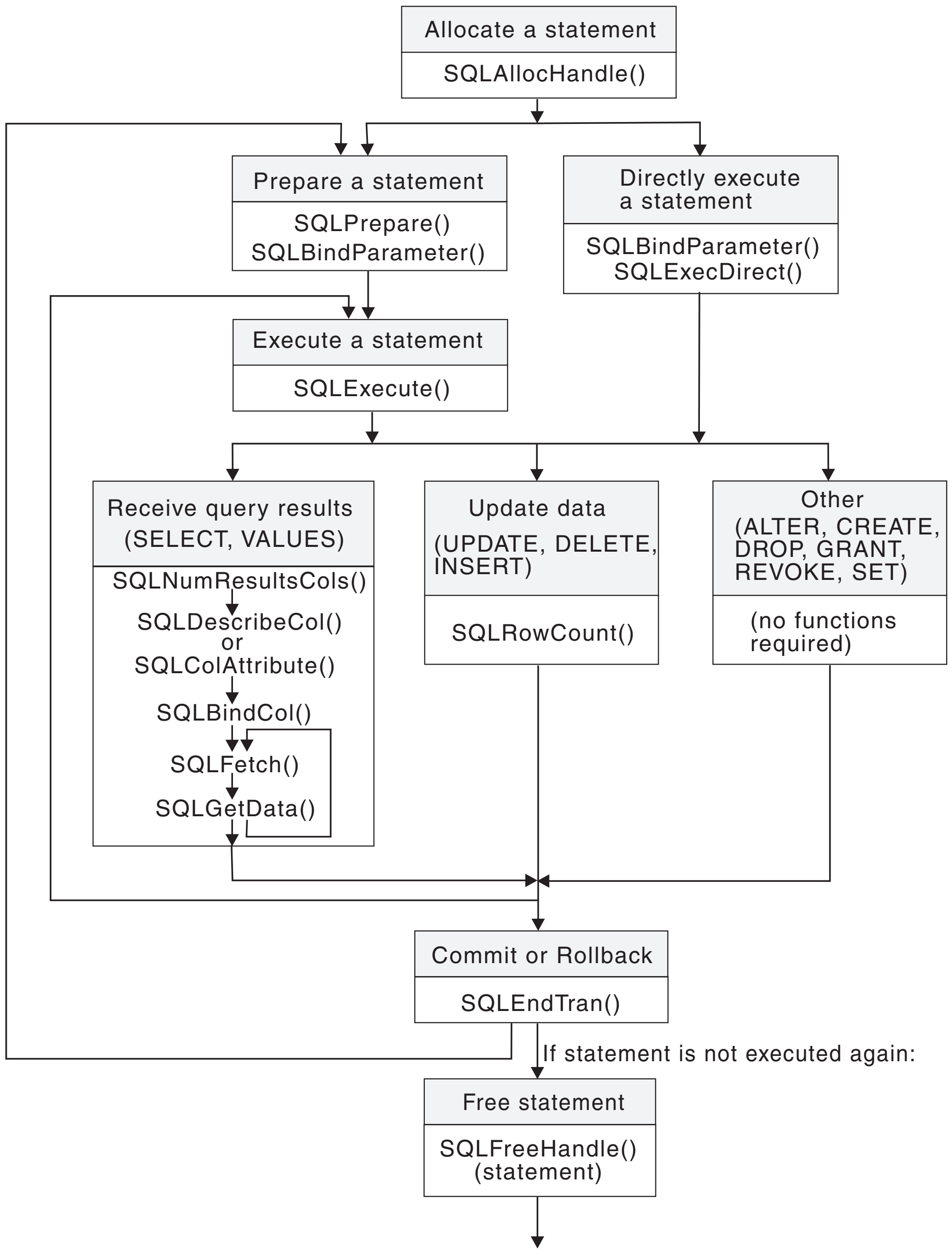


Figure 4. Transaction processing

Figure 4 shows the steps and the DB2 ODBC functions in the transaction processing task. This task contains five general steps:

1. Allocating statement handles
2. Preparing and executing SQL statements
3. Processing results
4. Committing or rolling back

5. Optionally, freeing statement handles if the statement is unlikely to be executed again

Allocating statement handles

SQLAllocHandle() (with *HandleType* set to SQL_HANDLE_STMT) allocates a statement handle. A statement handle refers to the data object that describes, and that tracks the execution of, an SQL statement. The description of an SQL statement 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 associates the statement it describes with a connection.

You must allocate a statement handle before you can execute a statement. By default, the maximum number of statement handles you can allocate at any one time is limited by the application heap size. The maximum number of statement handles you can actually use, however, is defined by DB2 ODBC. Table 2 lists the number of statement handles DB2 ODBC allows for each isolation level. If an application exceeds these limits, SQLPrepare() and SQLExecDirect() return SQLSTATE **HY**014.

Table 2. Maximum number of statement handles allocated at one time

Isolation level	Without hold	With hold	Total
Cursor stability	296	254	550
No commit	296	254	550
Repeatable read	296	254	550
Read stability	296	254	550
Uncommitted read	296	254	550

Preparation and execution

After you allocate a statement handle, you can specify and execute SQL statements with either of the two following methods:

- Prepare then execute, which consists of the following steps:
  1. Call SQLPrepare() with an SQL statement as an argument.
  2. Call SQLBindParameter() if the SQL statement contains *parameter markers*.
  3. Call SQLExecute().
- Execute direct, which consists of the following steps:
  1. Call SQLBindParameter() if the SQL statement contains *parameter markers*.
  2. Call SQLExecDirect() with an SQL statement as an argument.

The first method, prepare then execute, splits the preparation of the statement from the execution. Use this method when either of the following conditions is true:

- You execute a statement repeatedly (usually with different parameter values). This method allows you to prepare the same statement only once. Subsequent executions of that statement make use of the access plan the prepare generated.
- You require information about the columns in the result set, before it executes the statement.

The second method combines the prepare step and the execute step into one. Use this method when both of the following conditions are true:

- You execute the statement only once. This method allows you to call one function instead of two to execute an SQL statement.

- You do not require information about the columns in the result set before you actually execute the statement.

DB2 UDB for z/OS and DB2 UDB provide *dynamic statement caching* at the database server. In DB2 ODBC terms, *dynamic statement caching* means that for a given statement handle, once the database prepares a statement, it does not need to prepare it again (even after commits or rollbacks), as long as you do not free the statement handle. 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.

Binding parameters in SQL statements

Both SQLPrepare() followed by SQLExecute(), or SQLExecDirect() enable you to execute an SQL statement that uses parameter markers in place of expressions or host variables (for embedded SQL).

Parameter markers are question mark characters (?) that you place in SQL statements. When you execute a statement that contains parameter markers, these markers are replaced with the contents of host variables.

Binding associates an application variable to a parameter marker. Your application must bind an application variable to each parameter marker in an SQL statement before it can execute that statement. To bind a parameter, call SQLBindParameter() with the appropriate 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.

You refer to parameter markers in an SQL statement sequentially, from left to right, starting at 1, in ODBC function calls. You can call SQLNumParams() to determine the number of parameters in a statement. For more information about binding parameters in an SQL statement, see “SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator” on page 85.

The bound application variable and its associated length are called *deferred* input arguments. These arguments are called deferred because only pointers are passed when the parameter is bound; no data is read from the variable until the statement is executed. Deferred arguments enable you to modify the contents of bound parameter variables and execute SQL statements that use the most recent value with another call to SQLExecute().

Information for each parameter remains in effect until the application overrides or unbinds the parameter, or drops the statement handle. If the application executes the SQL statement repeatedly without changing the parameter binding, DB2 ODBC 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 deallocate or discard deferred input fields between the time it binds the fields to parameter markers and the time DB2 ODBC accesses them at execution time.

You can bind parameters to a variable with a different data type than the SQL statement requires. Your application must indicate the C data type of the source, and the SQL type of the parameter marker. DB2 ODBC converts the contents of the variable to match the SQL data type you specified. For example, the SQL statement might require an integer value, but your application has a string representation of an integer. You can bind the string to the parameter, and DB2 ODBC will convert the string to the corresponding integer value when you execute the statement. Not every C data type can be bound to a parameter marker.

Use SQLDescribeParam() to determine the data type of a parameter marker. If the application indicates an incorrect data type for the parameter marker, an extra conversion by the DBMS or an error can occur.

See "Data types and data conversion" on page 24 for more information about data conversion.

When you use an SQL predicate that compares a distinct type to a parameter marker, you must either cast the parameter marker to the distinct type or cast the distinct type to a source type. Otherwise, an error occurs. For an example of casting distinct types, see "Distinct types" on page 471.

For information about more advanced methods of binding application storage to parameter markers, see "Using arrays to pass parameter values" on page 414 and "Sending or retrieving long data values in pieces" on page 412.

Processing results

After an application executes an SQL statement, it must process the results that statement produced. The type of processing an application must employ depends on the type of SQL statement that it initially issued.

Processing query (SELECT, VALUES) statements

If an application issues a query statement, it can retrieve each row of the result set with the following steps:

1. Establish (describe) the structure of the result set, the number of columns, the column types, and the column 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 5, "Using advanced features," on page 397 discusses advanced techniques of using SQLExtendedFetch() to fetch multiple rows at a time.

Step 1

Analyze the executed or prepared statement. If the SQL statement was generated by the application, then this step might not be necessary because the application might know the structure of the result set and the data types of each column.

If you know the structure of the entire result set, especially if the result set contains a very large number of columns, you might want to supply DB2 ODBC with the descriptor information. This can reduce network traffic because DB2 ODBC does not have to retrieve the information from the server.

If the SQL statement was generated at run time (for example, entered by a user), then the application has 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 by calling SQLColAttribute(), after preparing or after executing the statement.

Step 2

The application retrieves 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 can 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 deallocate or discard variables used for deferred output fields between the time it binds them to columns of the result set and the time DB2 ODBC writes the data to these fields.

Step 3

Call SQLFetch() to fetch the first or next row of the result set. If any columns are bound, the application variable is updated. You can also write an application that fetches multiple rows of the result set into an array. See “Retrieving a result set into an array” on page 417 for more information.

If data conversion was indicated by the data types specified on the call to SQLBindCol(), the conversion occurs when SQLFetch() is called. See “Data types and data conversion” on page 24 for an explanation.

Step 4 (optional)

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. See “Data types and data conversion” on page 24 for more information.

To unbind a particular column of the result set, use SQLBindCol() with a null pointer for the application variable argument (*rgbValue*). To unbind all of the columns at one time, call SQLFreeHandle() on the statement handle.

Applications generally perform better if columns are bound rather than retrieved using SQLGetData(). However, an application can be constrained in the amount of long data that it can retrieve and handle at one time. If this is a concern, then SQLGetData() might be the better choice.

For information about more advanced methods for binding application storage to result set columns, see “Retrieving a result set into an array” on page 417 and “Sending or retrieving long data values in pieces” on page 412.

Processing UPDATE, DELETE and INSERT statements

If a statement modifies data (UPDATE, DELETE or INSERT statements), no action is required, other than the normal check for diagnostic messages. In this case, use SQLRowCount() to obtain the number of rows the SQL statement affects.

If the SQL statement is a positioned UPDATE or DELETE, you need 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 read-only.) In embedded SQL, the names of cursors are used to retrieve, update or delete rows. In DB2 ODBC, a cursor name is needed only for positioned UPDATE or DELETE SQL statements as they reference the cursor by name.

To perform a positioned update or delete in your application, use the following procedure:

1. Issue a SELECT statement to generate a result set.
2. Call SQLGetCursorName() to retrieve the name of the cursor on the result set that you generate in step 1. You use this cursor name in the UPDATE or DELETE statement.

   Tip: Use the name that DB2 automatically generates. Although you can define your own cursor names by using SQLSetCursorName(), use the name that DB2 generates. All error messages reference the DB2 generated name, not the name that you define with SQLSetCursorName().
3. Allocate a second statement handle to execute the positioned update or delete.

   To update or delete a row that has been fetched, you use two statement handles: one handle for the fetch and one handle for the update of the delete. You cannot reuse the fetch statement handle to execute a positioned update or delete because this handle holds the cursor while the positioned update or delete executes.
4. Call SQLFetch() to position the cursor on a row in the result set.
5. Create the UPDATE or DELETE SQL statement with the WHERE CURRENT of clause and specify the cursor name that you obtained in step 2.

   Example:

   ```
   sprintf((char *)stmtPositionedUpdate,
   "UPDATE org SET location = 'San Jose'  WHERE CURRENT of %s",
    cursorName);
   ```
6. Execute the positioned update or delete statement.

Processing other statements

If the statement neither queries nor modifies data, take no further action other than a 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 ODBC supports two commit modes: *autocommit* and *manual-commit*.

In autocommit mode, every SQL statement is a complete transaction, which is automatically committed. For a non-query statement, the commit is issued at the end of statement execution. For a query statement, the commit is issued after the cursor is closed. Given a single statement handle, the application must not start a second query before the cursor of the first query is closed.

In manual-commit mode, transactions are started implicitly with the first access to the data source using SQLPrepare(), SQLExecDirect(), SQLGetTypeInfo(), or any function that returns a result set, such as those described in "Querying catalog information" on page 407. At this point a transaction begins, 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 autocommit (except when participating in a coordinated transaction, see "Using a distributed unit of work" on page 399). An application can switch between manual-commit and autocommit modes by calling SQLSetConnectAttr(). Typically, a query-only application might wish to stay in autocommit mode. Applications that need to perform updates to the data source should turn off autocommit as soon as the data source connection is established.

When multiple connections exist, each connection has its own transaction (unless CONNECT (type 2) is specified). 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. Unlike distributed unit of work connections (described in "Using a distributed unit of work" on page 399), transactions on each connection do not coordinate.

When to call SQLEndTran()

If the application is in autocommit 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 can apply. See "Using a distributed unit of work" on page 399 for details.

Recommendation: If your application performs updates, do not wait until the application disconnects before you commit or roll back transactions.

The other extreme is to operate in autocommit mode, which is also not recommended as this adds extra processing. The application can modify the autocommit mode by invoking the SQLSetConnectAttr() function. See "Setting and retrieving environment, connection, and statement attributes" on page 397 and the SQLSetConnectAttr() function for information about switching between autocommit and manual-commit.

Consider the following behaviors to decide where in the application to end a transaction:

- If using CONNECT (type 1) with MULTICONTEXT=0, only the current connection can have an outstanding transaction. If using CONNECT (type 2), all connections participate in a single transaction.
- If using MULTICONTEXT=1, each connection can have an outstanding transaction.
- Various resources can be held while you have an outstanding transaction. Ending the transaction releases the resources for use by other users.

- When a transaction is successfully 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, an application behaves with the following characteristics:

- 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 if the data source supports statement caching (DB2 UDB for z/OS does). After a statement is prepared on a specific statement handle, it does not need to be prepared again 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). All cursors are defined using the WITH HOLD clause (except when connected to DB2 Server for VSE & VM, which does not support the WITH HOLD clause). For information about changing the default behavior, see "SQLSetStmtOption() - Set statement attribute" on page 375.

For more information and an example see "SQLTransact() - Transaction management" on page 396.

Freeing statement handles

Call SQLFreeHandle() (with *HandleType* set to SQL_HANDLE_STMT) to end processing for a particular statement handle. This function also performs the following tasks:

- Unbinds all columns of the result set
- Unbinds all parameter markers
- Closes any cursors and discard any pending results
- Drops the statement handle, and release all associated resources

The statement handle can be reused for other statements provided it is not dropped. If a statement handle is reused for another SQL statement string, any cached access plan for the original statement is discarded.

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 might occur.

Diagnostics

Diagnostics refers to dealing with warning or error conditions generated within an application. DB2 ODBC functions generate the following two levels of diagnostics:

- Return codes
- Detailed diagnostics (SQLSTATEs, messages, SQLCA)

Each DB2 ODBC function returns the function return code as a basic diagnostic. The SQLGetDiagRec() function provides 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, and the SQLSTATEs allow determination of the specific causes of failure.

The SQLGetDiagRec() function returns the following 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 an example, see “SQLGetDiagRec() - Get multiple field settings of diagnostic record” on page 221.

For diagnostic information about DB2 ODBC traces and debugging, see Chapter 6, “Problem diagnosis,” on page 477.

Function return codes

Table 3 lists all possible return codes for DB2 ODBC functions.

Table 3. DB2 ODBC 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 class is ’**01**’. See Table 233 on page 497.
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 might be available which can be obtained by calling SQLGetDiagRec().
SQL_NEED_DATA	The application tried to execute an SQL statement but DB2 ODBC lacks parameter data that the application had indicated would be passed at execute time. For more information, see “Sending or retrieving long data values in pieces” on page 412.
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

Because different database servers often have different diagnostic message codes, DB2 ODBC provides a standard set of codes, which are called *SQLSTATEs*. SQLSTATEs are defined by the X/Open SQL CAE specification. This allows consistent message handling across different database servers.

SQLSTATEs are alphanumeric strings of five characters (bytes) with a format of ***cc****sss*, where ***cc*** indicates class and *sss* indicates subclass. All SQLSTATEs use one of the following classes:

- ’**01**’, which is a warning
- ’**S1**’, which is generated by the DB2 ODBC driver for ODBC 2.0 applications
- ’**HY**’, which is generated by the DB2 ODBC driver for ODBC 3.0 applications

Important: In ODBC 3.0, '**HY**' classes map to '**S1**' classes. '**HY**' is a reserved X/Open class for ODBC/CLI implementations. This class replaces the '**S1**' class in ODBC 3.0 to follow the X/Open and/or ISO CLI standard. See "SQLSTATE mappings" on page 528 for more information.

For some error conditions, DB2 ODBC returns SQLSTATEs that differ from those states listed in the *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*. This inconsistency is a result of DB2 ODBC following the X/Open SQL CAE and SQL92 specifications.

DB2 ODBC SQLSTATEs include both additional IBM-defined SQLSTATEs that are returned by the database server, and DB2 ODBC-defined SQLSTATEs for conditions that are not defined in the X/Open specification. This allows for the maximum amount of diagnostic information to be returned.

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 ODBC SQLSTATEs that are defined by the X/Open specification, and return the additional ones as information only. (Dependencies refers to the application making logic flow decisions based on specific SQLSTATEs.)

  Tip: Consider building dependencies on the class (the first two characters) of the SQLSTATEs.
- For maximum diagnostic information, return the text message along with the SQLSTATE (if applicable, the text message also includes the IBM-defined SQLSTATE). It is also useful for the application to print out the name of the function that returned the error.

See Table 233 on page 497 for a listing and description of the SQLSTATEs explicitly returned by DB2 ODBC.

SQLCA

Embedded applications rely on the SQLCA for all diagnostic information. Although DB2 ODBC applications can retrieve much of the same information by using SQLGetDiagRec(), the application might still need to access the SQLCA that is related to the processing of a statement. (For example, after preparing a statement, the SQLCA contains the relative cost of executing the statement.) The SQLCA contains meaningful information only after 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. See "SQLGetSQLCA() - Get SQLCA data structure" on page 265 for more information.

Data types and data conversion

When you write a DB2 ODBC application, you must work with both SQL data types and C data types. Using both of these data types is unavoidable because the DBMS uses SQL data types, and the application uses 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 ODBC functions).

To help address this, DB2 ODBC provides symbolic names for the various data types, and manages the transfer of data between the DBMS and the application. It also performs data conversion (from a C character string to an SQL INTEGER type, for example) if required. To accomplish this, DB2 ODBC 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

These data types represent the combination of the ODBC 3.0 minimum, core, and extended data types. DB2 ODBC supports the following additional data types:

- SQL_GRAPHIC
- SQL_VARGRAPHIC
- SQL_LONGVARGRAPHIC

Table 4 lists each of the SQL data types, with its corresponding symbolic name, and the default C symbolic name. The table contains the following columns:

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 SQL symbolic names that are defined (in sqlcli1.h) as an integer value. These values are used by various functions to identify the SQL data types listed in the first column. See "Example" on page 133 for an example using these values.

Default C symbolic data type
This column contains C symbolic names, also defined as integer values. These values are used in various function arguments to identify the C data type as shown in Table 5 on page 27. The symbolic names are used by various functions (such as SQLBindParameter(), SQLGetData(), and SQLBindCol() calls) 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 ODBC assumes 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 4. SQL symbolic and default data types

SQL data type	Symbolic SQL data type	Default symbolic C data type
BLOB	SQL_BLOB	SQL_C_BINARY
BLOB LOCATOR[1]	SQL_BLOB_LOCATOR	SQL_C_BLOB_LOCATOR
CHAR	SQL_CHAR	SQL_C_CHAR
CHAR FOR BIT DATA	SQL_BINARY	SQL_C_BINARY
CLOB	SQL_CLOB	SQL_C_CHAR
CLOB LOCATOR	SQL_CLOB_LOCATOR	SQL_C_CLOB_LOCATOR
DATE	SQL_TYPE_DATE[2]	SQL_C_TYPE_DATE[2]
DBCLOB	SQL_DBCLOB	SQL_C_DBCHAR
DBCLOB LOCATOR[1]	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 or SQL_C_WCHAR[3]

Table 4. SQL symbolic and default data types (continued)

SQL data type	Symbolic SQL data type	Default symbolic C data type
INTEGER	SQL_INTEGER	SQL_C_LONG
LONG VARCHAR[4]	SQL_LONGVARCHAR	SQL_C_CHAR
LONG VARCHAR FOR BIT DATA[4]	SQL_LONGVARBINARY	SQL_C_BINARY
LONG VARGRAPHIC[4]	SQL_LONGVARGRAPHIC	SQL_C_DBCHAR or SQL_C_WCHAR[3]
NUMERIC[5]	SQL_NUMERIC[5]	SQL_C_CHAR
REAL[6]	SQL_REAL	SQL_C_FLOAT
ROWID	SQL_ROWID	SQL_C_CHAR
SMALLINT	SQL_SMALLINT	SQL_C_SHORT
TIME	SQL_TYPE_TIME[2]	SQL_C_TYPE_TIME[2]
TIMESTAMP	SQL_TYPE_TIMESTAMP[2]	SQL_C_TYPE_TIMESTAMP[2]
VARCHAR	SQL_VARCHAR	SQL_C_CHAR
VARCHAR FOR BIT DATA[4]	SQL_VARBINARY	SQL_C_BINARY
VARGRAPHIC	SQL_VARGRAPHIC	SQL_C_DBCHAR or SQL_C_WCHAR[3]

Notes:

1. LOB locator types are not persistent SQL data types (columns cannot be defined by a locator type; instead, it describes parameter markers, or represents a LOB value). See "Using large objects" on page 423 for more information.
2. See "Changes to datetime data types" on page 529 for information about data types used in previous releases.
3. The default C data type conversion for this SQL data type depends upon the encoding scheme your application uses. If your application uses UCS-2 Unicode encoding, the default conversion is to SQL_C_WCHAR. For all other encoding schemes the default conversion is to SQL_C_DBCHAR. See "Handling application encoding schemes" on page 443 for more information.
4. Whenever possible, replace long data types and FOR BIT DATA data types with appropriate LOB types.
5. NUMERIC is a synonym for DECIMAL on DB2 UDB for z/OS, DB2 for VSE & VM and DB2 UDB.
6. REAL is not valid for DB2 UDB or DB2 UDB for z/OS.

Additional information:

- DB2 ODBC for z/OS does not support the extended SQL data type SQL_BIGINT because DB2 UDB for z/OS does not support the data type BIGINT. On other DB2 platforms, which support the BIGINT data type, DB2 ODBC supports SQL_BIGINT.
- The data types, DATE, DECIMAL, NUMERIC, TIME, and TIMESTAMP cannot be transferred to their default C buffer types without a conversion.

Table 5 on page 27 shows the generic C type definitions for each symbolic C type. The table contains the following columns:

C symbolic data type
This column contains C symbolic names, defined as integer values. These values are used in various function arguments to identify the C data type shown in the last column. See "Example" on page 84 for an example using these values.

C type
This column contains C-defined types, which are defined in sqlcli1.h using a C typedef statement. The values in this column should be used to declare all DB2 ODBC related variables and arguments, in order to make the application more portable. See Table 7 on page 28 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 6 on page 27.

Table 5. 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_TYPE_DATE[1]	DATE_STRUCT	See Table 6
SQL_C_TYPE_TIME[1]	TIME_STRUCT	See Table 6
SQL_C_TYPE_TIMESTAMP[1]	TIMESTAMP_STRUCT	See Table 6
SQL_C_CLOB_LOCATOR	SQLINTEGER	Long int
SQL_C_BINARY	SQLCHAR	Unsigned char
SQL_C_BLOB_LOCATOR	SQLINTEGER	Long int
SQL_C_DBCHAR	SQLDBCHAR	Unsigned short
SQL_C_DBCLOB_LOCATOR	SQLINTEGER	Long int
SQL_C_WCHAR	SQLWCHAR	wchar_t

Note:

1. See "Changes to datetime data types" on page 529 for information about data types used in previous releases.

Table 6 lists the C data types with their associated structures for date, time and timestamp.

Table 6. C date, time, and timestamp structures

C type	Generic structure

DATE_STRUCT	`typedef struct DATE_STRUCT`
	`  {`
	`    SQLSMALLINT    year;`
	`    SQLUSMALLINT   month;`
	`    SQLUSMALLINT   day;`
	`  } DATE_STRUCT;`

TIME_STRUCT	`typedef struct TIME_STRUCT`
	`  {`
	`    SQLUSMALLINT   hour;`
	`    SQLUSMALLINT   minute;`
	`    SQLUSMALLINT   second;`
	`  } TIME_STRUCT;`

Table 6. C date, time, and timestamp structures (continued)

C type	Generic structure
TIMESTAMP_STRUCT	`typedef struct TIMESTAMP_STRUCT { SQLUSMALLINT year; SQLUSMALLINT month; SQLUSMALLINT day; SQLUSMALLINT hour; SQLUSMALLINT minute; SQLUSMALLINT second; SQLINTEGER fraction; } TIMESTAMP_STRUCT;`

```
typedef struct TIMESTAMP_STRUCT
  {
    SQLUSMALLINT   year;
    SQLUSMALLINT   month;
    SQLUSMALLINT   day;
    SQLUSMALLINT   hour;
    SQLUSMALLINT   minute;
    SQLUSMALLINT   second;
    SQLINTEGER    fraction;
  } TIMESTAMP_STRUCT;
```

See Table 7 for more information about the SQLUSMALLINT C data type.

Other C data types

In addition to the data types that map to SQL data types, other C symbolic types are used for other function arguments, such as pointers and handles. Table 7 shows both generic and ODBC data types used for these arguments.

Table 7. 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.
SQLHENV	long int	Handle referencing environment information.
SQLHDBC	long int	Handle referencing data source 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 ODBC functions.
SQLWCHAR	wchar_t	Data type for a Unicode UCS-2 character.
SQLWCHAR *	wchar_t *	Pointer to storage for Unicode UCS-2 data.

Data conversion

DB2 ODBC 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 4 on page 25, to identify the data types involved in the data transfer.

Example: The following SQLBindParameter() call binds a parameter marker that corresponds to an SQL data type of DECIMAL(5,3) to an application's C buffer type of double:

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

Table 4 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 8 shows all the conversions that DB2 ODBC supports.

Table 8 and Table 9 on page 32 list the data conversions DB2 ODBC supports.

Table 8 lists the conversions by SQL type. The first column of this table contains these SQL types. The second column of this table contains the default C type that the SQL type is converted to when you specify SQL_C_DEFAULT as the target type. The last column lists all other C types that you can specify as a target in a conversion from SQL data types to C data types.

Table 8. Supported data conversions by SQL data type

SQL symbolic data type	Default C symbolic data type	Additional C symbolic data types
SQL_BLOB	SQL_C_BINARY	SQL_C_CHAR[1] SQL_C_WCHAR[2] SQL_C_BLOB_LOCATOR[3]
SQL_CHAR	SQL_C_CHAR[1]	SQL_C_WCHAR[2] 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_BLOB_LOCATOR[3]
SQL_CLOB	SQL_C_CHAR[1]	SQL_C_WCHAR[2] SQL_C_BINARY SQL_C_CLOB_LOCATOR[3]
SQL_DBCLOB	SQL_C_DBCHAR	SQL_C_BINARY SQL_C_DBCLOB_LOCATOR[3]
SQL_DECIMAL	SQL_C_CHAR[1]	SQL_C_WCHAR[2] SQL_C_LONG SQL_C_SHORT SQL_C_TINYINT SQL_C_FLOAT SQL_C_DOUBLE SQL_C_BINARY SQL_C_BIT
SQL_DOUBLE	SQL_C_DOUBLE	SQL_C_CHAR[1] SQL_C_WCHAR[2] SQL_C_LONG SQL_C_SHORT SQL_C_TINYINT SQL_C_FLOAT SQL_C_BIT

Table 8. Supported data conversions by SQL data type (continued)

SQL symbolic data type	Default C symbolic data type	Additional C symbolic data types
SQL_FLOAT	SQL_C_DOUBLE	SQL_C_CHAR[1] SQL_C_WCHAR[2] SQL_C_LONG SQL_C_SHORT SQL_C_TINYINT SQL_C_FLOAT SQL_C_BIT
SQL_GRAPHIC	SQL_C_DBCHAR or SQL_C_WCHAR[4]	SQL_C_CHAR[1]
SQL_INTEGER	SQL_C_LONG	SQL_C_CHAR[1] SQL_C_WCHAR[2] SQL_C_SHORT SQL_C_TINYINT SQL_C_FLOAT SQL_C_DOUBLE SQL_C_BIT
SQL_LONGVARCHAR	SQL_C_CHAR[1]	SQL_C_WCHAR[2] SQL_C_TYPE_DATE SQL_C_TYPE_TIMESTAMP SQL_C_BINARY
SQL_LONGVARGRAPHIC	SQL_C_DBCHAR or SQL_C_WCHAR[4]	SQL_C_CHAR[1] SQL_C_BINARY
SQL_NUMERIC[5]	SQL_C_CHAR[1]	SQL_C_WCHAR[2] SQL_C_LONG SQL_C_SHORT SQL_C_TINYINT SQL_C_FLOAT SQL_C_DOUBLE SQL_C_BIT
SQL_REAL[6]	SQL_C_FLOAT	SQL_C_CHAR[1] SQL_C_WCHAR[2] SQL_C_LONG SQL_C_SHORT SQL_C_TINYINT SQL_C_DOUBLE SQL_C_BIT
SQL_ROWID	SQL_C_CHAR	SQL_C_WCHAR
SQL_SMALLINT	SQL_C_SHORT	SQL_C_CHAR[1] SQL_C_WCHAR[2] SQL_C_LONG 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[3] SQL_C_BLOB_LOCATOR[3] SQL_C_DBCLOB_LOCATOR[3]

Table 8. Supported data conversions by SQL data type (continued)

SQL symbolic data type	Default C symbolic data type	Additional C symbolic data types
SQL_TYPE_DATE	SQL_C_TYPE_DATE	SQL_C_CHAR[1] SQL_C_WCHAR[2] SQL_C_TYPE_TIMESTAMP
SQL_TYPE_TIME	SQL_C_TYPE_TIME	SQL_C_CHAR[1] SQL_C_WCHAR SQL_C_TYPE_TIMESTAMP
SQL_TYPE_TIMESTAMP	SQL_C_TYPE_TIMESTAMP	SQL_C_CHAR[1] SQL_C_WCHAR[2] SQL_C_TYPE_DATE SQL_C_TYPE_TIME
SQL_VARCHAR	SQL_C_CHAR[1]	SQL_C_WCHAR[2] 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_BINARY SQL_C_BIT SQL_C_TYPE_TIMESTAMP
SQL_VARGRAPHIC	SQL_C_DBCHAR or SQL_C_WCHAR[4]	SQL_C_CHAR[1]

Notes:

1. You must bind data to the SQL_C_CHAR data type for Unicode UTF-8 data
2. You must bind data with the SQL_C_WCHAR data type for Unicode UCS-2 data.
3. Data is not converted to LOB locator types; locators represent a data value.
4. The default C data type conversion for this SQL data type depends upon the encoding scheme your application uses. If your application uses UCS-2 Unicode encoding, the default conversion is to SQL_C_WCHAR. For all other encoding schemes the default conversion is to SQL_C_DBCHAR. See "Handling application encoding schemes" on page 443 for more information.
5. NUMERIC is a synonym for DECIMAL on DB2 UDB for z/OS, DB2 for VSE & VM, and DB2 UDB.
6. REAL is not supported by DB2 UDB except in the z/OS environment.

Table 9 on page 32 lists the conversions by C type. The first column of this table contains these C types. The second column of this table contains the SQL types that use the C type in the first column for default conversions. The last column are all other SQL types you can specify in a conversion from C data types to SQL data types.

Table 9. Supported data conversions by C data type

Symbolic C data type	Symbolic SQL data types that use this C data type as a default	Additional symbolic SQL data types
SQL_C_CHAR[1]	SQL_CHAR SQL_CLOB SQL_DECIMAL SQL_LONGVARCHAR SQL_NUMERIC[2] SQL_VARCHAR	SQL_BLOB SQL_DOUBLE SQL_FLOAT SQL_GRAPHIC SQL_INTEGER SQL_LONGVARGRAPHIC SQL_REAL[3] SQL_ROWID SQL_SMALLINT SQL_TYPE_DATE SQL_TYPE_TIME SQL_TYPE_TIMESTAMP SQL_VARGRAPHIC
SQL_C_WCHAR[4]	SQL_GRAPHIC[6] SQL_LONGVARGRAPHIC[6] SQL_VARGRAPHIC[6]	SQL_BLOB SQL_CHAR SQL_CLOB SQL_DECIMAL SQL_DOUBLE SQL_FLOAT SQL_INTEGER SQL_LONGVARCHAR SQL_NUMERIC[2] SQL_REAL[3] SQL_ROWID SQL_SMALLINT SQL_TYPE_DATE SQL_TYPE_TIME SQL_TYPE_TIMESTAMP SQL_VARCHAR
SQL_C_LONG	SQL_INTEGER	SQL_CHAR SQL_DECIMAL SQL_DOUBLE SQL_FLOAT SQL_NUMERIC[2] SQL_REAL[3] SQL_SMALLINT SQL_VARCHAR
SQL_C_SHORT	SQL_SMALLINT	SQL_CHAR SQL_DECIMAL SQL_DOUBLE SQL_FLOAT SQL_INTEGER SQL_NUMERIC[3] SQL_REAL[4] SQL_VARCHAR
SQL_C_TINYINT	No SQL data types use SQL_C_TINYINT in a default conversion.	SQL_CHAR SQL_DECIMAL SQL_DOUBLE SQL_FLOAT SQL_INTEGER SQL_NUMERIC[2] SQL_REAL[3] SQL_SMALLINT SQL_VARCHAR

Table 9. *Supported data conversions by C data type (continued)*

Symbolic C data type	**Symbolic SQL data types that use this C data type as a default**	**Additional symbolic SQL data types**
SQL_C_FLOAT	SQL_REAL[3]	SQL_CHAR SQL_DECIMAL SQL_DOUBLE SQL_FLOAT SQL_INTEGER SQL_NUMERIC[2] SQL_SMALLINT SQL_VARCHAR
SQL_C_DOUBLE	SQL_DOUBLE SQL_FLOAT	SQL_CHAR SQL_DECIMAL SQL_INTEGER SQL_NUMERIC[2] SQL_REAL[3] SQL_SMALLINT SQL_VARCHAR
SQL_C_TYPE_DATE	SQL_TYPE_DATE	SQL_CHAR SQL_LONGVARCHAR SQL_TYPE_TIMESTAMP SQL_VARCHAR
SQL_C_TYPE_TIME	SQL_TYPE_TIME	SQL_CHAR SQL_TYPE_TIMESTAMP SQL_VARCHAR
SQL_C_TYPE_TIMESTAMP	SQL_TYPE_TIMESTAMP	SQL_CHAR SQL_LONGVARCHAR SQL_TYPE_DATE SQL_TYPE_TIME SQL_VARCHAR
SQL_C_BINARY	SQL_BLOB	SQL_CHAR SQL_CLOB SQL_DBCLOB SQL_DECIMAL SQL_LONGVARCHAR SQL_LONGVARGRAPHIC SQL_VARCHAR
SQL_C_BIT	No SQL types use SQL_C_BIT in a default conversion.	SQL_CHAR SQL_DECIMAL SQL_DOUBLE SQL_FLOAT SQL_INTEGER SQL_NUMERIC[2] SQL_REAL[3] SQL_SMALLINT SQL_VARCHAR
SQL_C_DBCHAR	SQL_DBCLOB SQL_GRAPHIC[6] SQL_LONGVARGRAPHIC[6] SQL_VARGRAPHIC[6]	No additional SQL data types can use SQL_C_DBCHAR.
SQL_C_CLOB_LOCATOR	No SQL data types use SQL_C_CLOB_LOCATOR in a default conversion.	SQL_CLOB

Table 9. Supported data conversions by C data type (continued)

Symbolic C data type	Symbolic SQL data types that use this C data type as a default	Additional symbolic SQL data types
| SQL_C_BLOB_LOCATOR	No SQL data types use SQL_C_BLOB_LOCATOR in a default conversion.	SQL_BLOB
| SQL_C_DBCLOB_LOCATOR	No SQL data types use SQL_C_DBCLOB_LOCATOR in a default conversion.	SQL_DBCLOB

Notes:

1. You must bind data to the SQL_C_CHAR data type for Unicode UTF-8 data
2. NUMERIC is a synonym for DECIMAL on DB2 UDB for z/OS, DB2 for VSE & VM, and DB2 UDB.
3. REAL is not supported by DB2 UDB.
4. You must bind data with the SQL_C_WCHAR data type for Unicode UCS-2 data.
5. Data is not converted to LOB locator types; locators represent a data value.
6. The default C data type conversion for this SQL data type depends upon the encoding scheme your application uses. If your application uses UCS-2 Unicode encoding, the default conversion is to SQL_C_WCHAR. For all other encoding schemes the default conversion is to SQL_C_DBCHAR. See "Handling application encoding schemes" on page 443 for more information.

See Appendix D, "Data conversion," on page 509 for information about required formats and the results of converting between data types.

Limits on precision, and scale, as well as truncation and rounding rules for type conversions follow rules specified in the *IBM SQL Reference* with the following exception; truncation of values to the right of the decimal point for numeric values returns 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 about the failure. When moving and converting floating point data values between the application and DB2 ODBC, no correspondence is guaranteed to be exact as the values can change in precision and scale.

Working with string arguments

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

Length of string arguments

Input string arguments have an associated length argument. This argument passes DB2 ODBC one of the following types of information:

- The exact length of the string (not including the nul-terminator)
- The special value SQL_NTS to indicate a nul-terminated string
- SQL_NULL_DATA to pass a null value

If the length is set to SQL_NTS, DB2 ODBC determines the length of the string by locating the nul-terminator. All length arguments for input/output strings are passed as a count of characters. Length arguments that can refer to both string and non-string data are passed as a count of bytes.

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 ODBC. 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.

If a function is called with a null pointer for an output length argument, DB2 ODBC does 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 ODBC 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.

Recommendation: Always use a valid output length argument.

If the length argument (*pcbValue*) and the output buffer (*rgbValue*) are contiguous in memory, DB2 ODBC can return both values more efficiently, improving application performance. For example, if the following structure is defined and *&buffer.pcbValue* and *buffer.rgbValue* are passed to SQLBindCol(), DB2 ODBC updates both values in one operation.

```
struct
{   SQLINTEGER pcbValue;
    SQLCHAR    rgbValue [BUFFER_SIZE];
} buffer;
```

Nul-termination of strings

By default, every character string that DB2 ODBC returns is terminated with a nul-terminator (hex 00), except for strings returned from the graphic and DBCLOB data types into SQL_C_CHAR application variables. Graphic and DBCLOB data types that are retrieved into SQL_C_DBCHAR and SQL_C_WCHAR application variables are nul-terminated with a double-byte nul-terminator (hex 0000). This requires that all buffers allocate enough space for the maximum number of bytes expected, plus the nul-terminator.

You can also use SQLSetEnvAttr() and set an environment attribute to disable nul-termination of varying-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 ODBC can indicate the actual length of data returned; otherwise, the application has no means to determine this. The DB2 ODBC default is to always write the nul-terminator.

String truncation

If an output string does not fit into a buffer, DB2 ODBC truncates the string to the size of the buffer, and writes the nul-terminator. If truncation occurs, the function returns SQL_SUCCESS_WITH_INFO and SQLSTATE **01**004, which indicates data 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.

ODBC specifies that string data can be truncated on input or output with the appropriate SQLSTATE. As the data source, DB2 does not truncate data on input, but might truncate data on output to maintain data integrity. On input, DB2 rejects string truncation with a negative SQLCODE (-302) and SQLSTATE **22**001. On output, DB2 truncates the data and issues SQL_SUCCESS_WITH_INFO and SQLSTATE **01**004.

Interpretation of strings

Normally, DB2 ODBC 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. In this case, if the cursor name is not delimited (enclosed by double quotes) the leading and trailing blanks are removed and case is preserved.

Querying environment and data source information

Many situations require an application retrieve information about the characteristics and capabilities of the current DB2 ODBC driver or the data source to which it is connected.

One of the most common situations involves displaying information for the user. Information such as the data source name and version, or the version of the DB2 ODBC driver might be displayed at connect time, or as part of the error reporting process.

These functions are also useful to generic applications that are written to adapt and take advantage of facilities that might be available from some, but not all database servers. The following DB2 ODBC functions provide data source specific information:
- "SQLDataSources() - Get a list of data sources" on page 127
- "SQLGetFunctions() - Get functions" on page 226
- "SQLGetInfo() - Get general information" on page 234
- "SQLGetTypeInfo() - Get data type information" on page 280

Example: Figure 5 on page 37 shows an application that queries an ODBC environment for a data source, all supported functions, and a supported data type.

```
/*********************************************************/
/*  Querying environment and data source information      */
/*********************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlcli1.h>

void main()
{
    SQLHENV         hEnv;           /* Environment handle              */
    SQLHDBC         hDbc;           /* Connection handle               */
    SQLRETURN       rc;             /* Return code for API calls       */
    SQLHSTMT        hStmt;          /* Statement handle                */
    SQLCHAR         dsname[30];     /* Data source name                */
    SQLCHAR         dsdescr[255];   /* Data source description         */
    SQLSMALLINT     dslen;          /* Length of data source           */
    SQLSMALLINT     desclen;        /* Length of dsdescr               */
    BOOL            found = FALSE;
    SQLSMALLINT     funcs[100];
    SQLINTEGER      rgbValue;

    /*
     *  Initialize environment - allocate environment handle.
     */
    rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv );
    rc = SQLAllocHandle( SQL_HANDLE_DBC, hEnv, &hDbc );

    /*
     *  Use SQLDataSources to verify MVSDB2 does exist.
     */
    while( ( rc = SQLDataSources( hEnv,
                        SQL_FETCH_NEXT,
                        dsname,
                        SQL_MAX_DSN_LENGTH+1,
                        &dslen,
                        dsdescr,
                        &desclen ) ) != SQL_NO_DATA_FOUND )
    {
       if( !strcmp( dsname, "MVSDB2" ) )   /* data source exist          */
       {
           found = TRUE;
           break;
       }
    }
```

Figure 5. An application that queries environment information. (Part 1 of 2)

```
    if( !found )
    {
        fprintf(stdout, "Data source %s does not exist...\n", dsname );
        fprintf(stdout, "program aborted.\n");
        exit(1);
    }
    if( ( rc = SQLConnect( hDbc, dsname, SQL_NTS, "myid", SQL_NTS,
                            "mypd", SQL_NTS ) )
         == SQL_SUCCESS )
    {
       fprintf( stdout, "Connect to %s\n", dsname );
    }

    SQLAllocHandle( SQL_HANDLE_STMT, hDbc, &hStmt );

    /*
     *   Use SQLGetFunctions to store all APIs status.
     */
    rc = SQLGetFunctions( hDbc, SQL_API_ALL_FUNCTIONS, funcs );

    /*
     *   Check whether SQLGetInfo is supported in this driver. If so,
     *   verify whether DATE is supported for this data source.
     */
    if( funcs[SQL_API_SQLGETINFO] == 1 )
    {
        SQLGetInfo( hDbc, SQL_CONVERT_DATE, (SQLPOINTER)&rgbValue, 255, &desclen );
        if( rgbValue & SQL_CVT_DATE )
        {
           SQLGetTypeInfo( hStmt, SQL_DATE );

           /*  use SQLBindCol and SQLFetch to retrieve data ....*/
        }
    }

}
```

Figure 5. An application that queries environment information. (Part 2 of 2)

Chapter 3. Configuring DB2 ODBC and running sample applications

This chapter provides information about installing DB2 ODBC, the DB2 ODBC run-time environment, and the preparation steps needed to run a DB2 ODBC application.

- "Installing DB2 ODBC"
- "The DB2 ODBC run-time environment"
- "Setting up the DB2 ODBC run-time environment" on page 41
- "Preparing and executing a DB2 ODBC application" on page 44
- "DB2 ODBC initialization file" on page 49
- "DB2 ODBC migration considerations" on page 62

Installing DB2 ODBC

You must edit and run SMP/E jobs to install DB2 ODBC. Section 2 of *DB2 Installation Guide* has information about the SMP/E jobs to receive, apply, and accept the FMIDs for DB2 ODBC. These jobs are run as part of the DB2 installation process.

1. Copy and edit the SMP/E jobs.

   For sample JCL to invoke the z/OS utility IEBCOPY to copy the SMP/E jobs to disk, see the *DB2 Program Directory*.
2. Run the receive job: DSNRECV3.
3. Run the apply job: DSNAPPLY.
4. Run the accept job: DSNACCEP.

Customize these jobs to specify data set names for your DB2 installation and SMP/E data sets. See the header notes in each job for details.

The DB2 ODBC run-time environment

DB2 ODBC does not support an ODBC driver manager. All API calls are routed through the single ODBC driver that is loaded at run time into the application address space. DB2 ODBC support is implemented as an IBM C/C++ Dynamic Load Library (DLL). By providing DB2 ODBC support using a DLL, DB2 ODBC applications do not need to link-edit any DB2 ODBC driver code with the application load module. Instead, the linkage to the DB2 ODBC APIs is resolved dynamically at run time by the IBM Language Environment® run-time support.

The DB2 ODBC driver can use either the call attachment facility (CAF) or the Resource Recovery Services attachment facility (RRSAF) to connect to the DB2 UDB for z/OS address space.

- If the DB2 ODBC application is **not** running as a DB2 UDB for z/OS stored procedure, the MVSATTACHTYPE keyword in the DB2 ODBC initialization file determines the attachment facility that DB2 ODBC uses.
- If the DB2 ODBC application is running as a DB2 UDB for z/OS stored procedure, then DB2 ODBC uses the attachment facility that was specified for stored procedures.

When the DB2 ODBC application invokes the first ODBC function, SQLAllocHandle() (with *HandleType* set to SQL_HANDLE_ENV), the DB2 ODBC driver DLL is loaded.

DB2 ODBC supports access to the local DB2 UDB for z/OS subsystems and any remote data source that is accessible using DB2 UDB for z/OS Version 8. This includes:
- Remote DB2 subsystems using specification of an alias or three-part name
- Remote DRDA-1 and DRDA-2 servers using LU 6.2 or TCP/IP.

The relationship between the application, the DB2 UDB for z/OS Version 8 ODBC driver and the DB2 UDB for z/OS subsystem are illustrated in Figure 6.

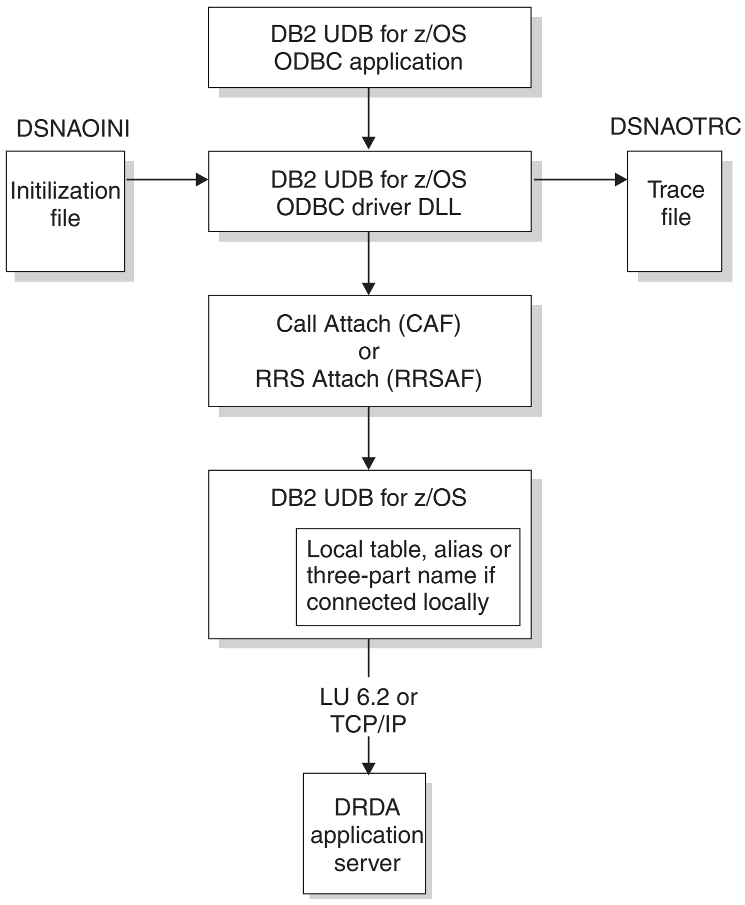

DB2 UDB for z/OS ODBC application
DSNAOINI
Initilization file
DB2 UDB for z/OS ODBC driver DLL
DSNAOTRC
Trace file
Call Attach (CAF) or RRS Attach (RRSAF)
DB2 UDB for z/OS
Local table, alias or three-part name if connected locally
LU 6.2 or TCP/IP
DRDA application server

Figure 6. Relationship between DB2 UDB for z/OS Version 8 ODBC components

Connectivity requirements

DB2 UDB for z/OS Version 8 ODBC has the following connectivity requirements:
- DB2 ODBC applications must execute on a machine on which Version 8 of DB2 UDB for z/OS is installed.
- If the application is executing with MULTICONTEXT=1, it can make multiple physical connections. Each connection corresponds to an independent transaction and DB2 thread.
- If the application is executing CONNECT (type 1) (described in “Specifying the connection type” on page 12) and MULTICONTEXT=0, only one current physical connection and one transaction on that connection occurs. All transactions on logical connections (that is, with a valid connection handle) are rolled back by the application or committed by DB2 ODBC. This is a deviation from the ODBC connection model.

Setting up the DB2 ODBC run-time environment

This section describes the general setup required to enable DB2 ODBC applications. The steps in this section only need to be performed once, and are usually performed as part of the installation process for DB2 UDB for z/OS.

The DB2 ODBC bind files must be bound to the data source. The following two bind steps are required:
- Create packages at every data source
- Create at least one plan to name those packages

These bind steps are described in the following sections:
- "Bind DBRMs to create packages"
- "Bind the application plan" on page 43

The online bind sample is available in DSN810.SDSNSAMP(DSNTIJCL). It is strongly recommended that you use this bind sample as a guide for binding DBRMs to packages and binding an application plan.

Special considerations for the z/OS UNIX environment are described in "Setting up the z/OS UNIX environment" on page 44

Bind DBRMs to create packages

This section explains how to bind database request modules (DBRMs) to create packages. Use the online bind sample, DSN810.SDSNSAMP(DSNTIJCL), for guidance.

For an application to access a data sources using DB2 ODBC, you must bind all required IBM DBRMs (which are shipped in DSN810.SDSNDBRM) to all data sources. These data sources include the local DB2 UDB for z/OS subsystem and all remote (DRDA) data sources. You can use the SQLConnect() argument *szDSN* to identify the data sources your DB2 ODBC applications access. The *szDSN* argument returns only data source names that appear in the DB2 SYSIBM.LOCATION catalog table. You do not need to bind an application that runs under DB2 ODBC and accesses remote data sources into the DB2 ODBC plan. You can bind applications as a package at the remote site. Failure to bind the package at the remote site results in SQLCODE -805.

- Bind the following DBRMs to all data sources using the isolation levels that are indicated:
  - **DSNCLICS** with ISOLATION(CS)
  - **DSNCLIRR** with ISOLATION(RR)
  - **DSNCLIRS** with ISOLATION(RS)
  - **DSNCLIUR** with ISOLATION(UR)
  - **DSNCLINC** with ISOLATION(NC)
- Bind the following DBRMs with default options to all OS/390 and z/OS servers:
  - **DSNCLIC1**
  - **DSNCLIC2**
  - **DSNCLIMS**
  - **DSNCLIF4**
- Bind **DSNCLIVM** with default options to DB2 for VSE & VM servers
- Bind **DSNCLIAS** with default options to DB2 UDB for iSeries servers

- Bind **DSNCLIV1** and **DSNCLIV2** with default options to all DB2 UDB for Linux, UNIX and Windows severs
- Bind **DSNCLIQR** to any site that supports DRDA query result sets

To call stored procedures that run under DB2 ODBC, bind each of these procedures to the data sources that use them. You do not need to bind a stored procedure that runs under DB2 ODBC into the DB2 ODBC plan. You can bind a stored procedure as a package. For more information about DB2 ODBC stored procedures, see "Using stored procedures" on page 429.

Package bind options

For packages listed above that use the ISOLATION keyword, the impact of package bind options in conjunction with the DB2 ODBC initialization file keywords is as follows:

- ISOLATION

  Packages must be bound with the isolation specified.
- DYNAMICRULES(BIND)

  Binding the packages with this option offers encapsulation and security similar to that of static SQL. The recommendations and consequences for using this option are as follows:
  1. Bind DB2 ODBC packages or plan with DYNAMICRULES(BIND) from a 'driver' authorization ID with table privileges.
  2. Issue GRANT EXECUTE on each collection or plan name to individual users. Packages are differentiated by collection; plans are differentiated by plan name.
  3. Select a plan or package by using the PLANNAME or COLLECTIONID keywords in the DB2 ODBC initialization file.
  4. When dynamic SQL is issued, the statement is processed with the 'driver' authorization ID. Users need execute privileges; table privileges are not required.
  5. The CURRENTSQLID keyword cannot be used in the DB2 ODBC initialization file. Use of this keyword results in an error at SQLConnect().
- ENCODING

  The ENCODING bind option controls the application encoding scheme for all static SQL statements in a plan or package.

  Important: You must specify ENCODING(EBCDIC) when you bind packages to the local DB2 UDB for z/OS subsystem.
- SQLERROR(CONTINUE)

  Use this keyword to bind DB2 ODBC to Version 5 of DB2 for OS/390. The symptoms of binding to a down-level server are:
  – Binding DSNCLIMS results in SQLCODE -199 on the VALUES INTO statement. Bind with the SQLERROR(CONTINUE) keyword to bypass this error.
  – Binding DSNCLIMS results in SQLCODE -199 on the DESCRIBE INPUT statement. Apply APAR PQ24584 and retry the bind to bypass this error. Alternatively, you can bind with the SQLERROR(CONTINUE) keyword, however, the SQLDescribeParam() API will be unavailable to you at that DB2 for OS/390 Version 5 server.
  – Binding DSNCLIMS on any DB2 subsystem that supports MIXED DATA results in SQLCODE -130. Bind with SQLERROR(CONTINUE) keyword to bypass this error.

Bind return codes

A bind to DB2 UDB for z/OS receives several expected warnings:

- For all packages:

  ```
  WARNING, ONLY IBM-SUPPLIED COLLECTION-IDS SHOULD BEGIN WITH "DSN"
  ```

- For bind of DSNCLINC package to DB2 UDB for z/OS:

  ```
  BIND WARNING - ISOLATION NC NOT SUPPORTED CHANGED TO ISOLATION UR
  ```

- For bind of DSNCLIF4 package to DB2 UDB for z/OS for SYSIBM.LOCATIONS due to differences in catalog table names between releases. For example, when bound to a Version 6 system you receive this warning message:

  ```
  SYSIBM.SYSLOCATIONS IS NOT DEFINED
  ```

Bind packages at remote sites

For an application to access a data source using DB2 ODBC, bind the DBRMs listed above to all data sources, including the local DB2 UDB for z/OS subsystem and all remote (DRDA) data sources. The SQLConnect() argument *szDSN* identifies the data source. The data source is the location in the DB2 SYSIBM.LOCATION catalog table. An application running under DB2 ODBC to a remote DB2 UDB for z/OS, or another DBMS, does not need to be bound into the DB2 ODBC plan; rather it can be bound as a package at the remote site. Failure to bind the package at the remote site results in SQLCODE -805.

Bind stored procedures

A stored procedure running under DB2 ODBC to a remote DB2 UDB for z/OS, or another DBMS, does not need to be bound into the DB2 ODBC plan; rather it can be bound as a package at the remote site.

For example, DB2 ODBC must always be bound in a plan to a DB2 UDB for z/OS subsystem to which DB2 ODBC first establishes an affinity on the SQLAllocHandle() call (with *HandleType* set to SQL_HANDLE_ENV). This is the local DB2. The scenario in this example is equivalent to specifying the MVSDEFAULTSSID keyword in the initialization file.

Bind the application plan

This section explains how to bind an application plan. Use the online bind sample, DSN810.SDSNSAMP(DSNTIJCL), for guidance.

A DB2 plan must be created using the PKLIST keyword to name all packages listed in “Bind DBRMs to create packages” on page 41. Any name can be selected for the plan; the default name is DSNACLI. If a name other than the default is selected, that name must be specified within the initialization file by using the PLANNAME keyword.

Plan bind options

Do not specify PLAN bind options when you bind the application plan. The bind options are used as follows:

- DISCONNECT(EXPLICIT)

  All DB2 ODBC plans are created using this option. DISCONNECT(EXPLICIT) is the default value; do not change it.

- CURRENTSERVER

  Do not specify this keyword when binding plans.

Setting up the z/OS UNIX environment

To use DB2 ODBC in the z/OS UNIX environment, the DB2 ODBC definition side-deck must be available to z/OS UNIX users.

The z/OS UNIX environment compiler determines the contents of an input file based on the file extension. In the case of a file residing in a partitioned data set (PDS), the last qualifier in the PDS name is treated as the file extension.

The z/OS UNIX environment compiler recognizes the DB2 ODBC definition side-deck by these criteria:

- It must reside in a PDS
- The last qualifier in the PDS name must be `.EXP`

Therefore, to make the DB2 ODBC definition side-deck available to z/OS UNIX environment users, you should define a data set alias that uses `.EXP` as the last qualifier in the name. This alias should relate to the SDSNMACS data set which is where the DB2 ODBC definition side-deck is installed.

For example, assume that DB2 is installed using DSN810 as the high level data set qualifier. You can define the alias using the following command:

```
DEFINE ALIAS(NAME('DSN810.SDSNC.EXP') RELATE('DSN810.SDSNMACS'))
```

This alias allows z/OS UNIX environment users to directly reference the DB2 ODBC definition side-deck by specifying the following input files as input to the z/OS UNIX environment `c89` command:

```
"//'DSN810.SDSNC.EXP(DSNAOCLI)'"
```

As an alternative to defining a system alias for the ODBC side-deck, use the *_XSUFFIX_HOST* environmental variable that specifies the z/OS data set suffix. The default value is EXP. For example, changing the default from EXP to SDSNMACS allows the link to work without a Define Alias.

For the c89 compiler, issue:

```
export _C89_XSUFFIX_HOST="SDSNMACS"
```

For the cxx compiler, issue:

```
export _CXX_XSUFFIX_HOST="SDSNMACS"
```

Preparing and executing a DB2 ODBC application

This section provides an overview of the DB2 ODBC components and explains the steps you follow to prepare and execute a DB2 ODBC application.

Figure 7 on page 45 shows the DB2 ODBC components that are used to build DB2 ODBC DLL. The shaded areas identify the components that are shipped.

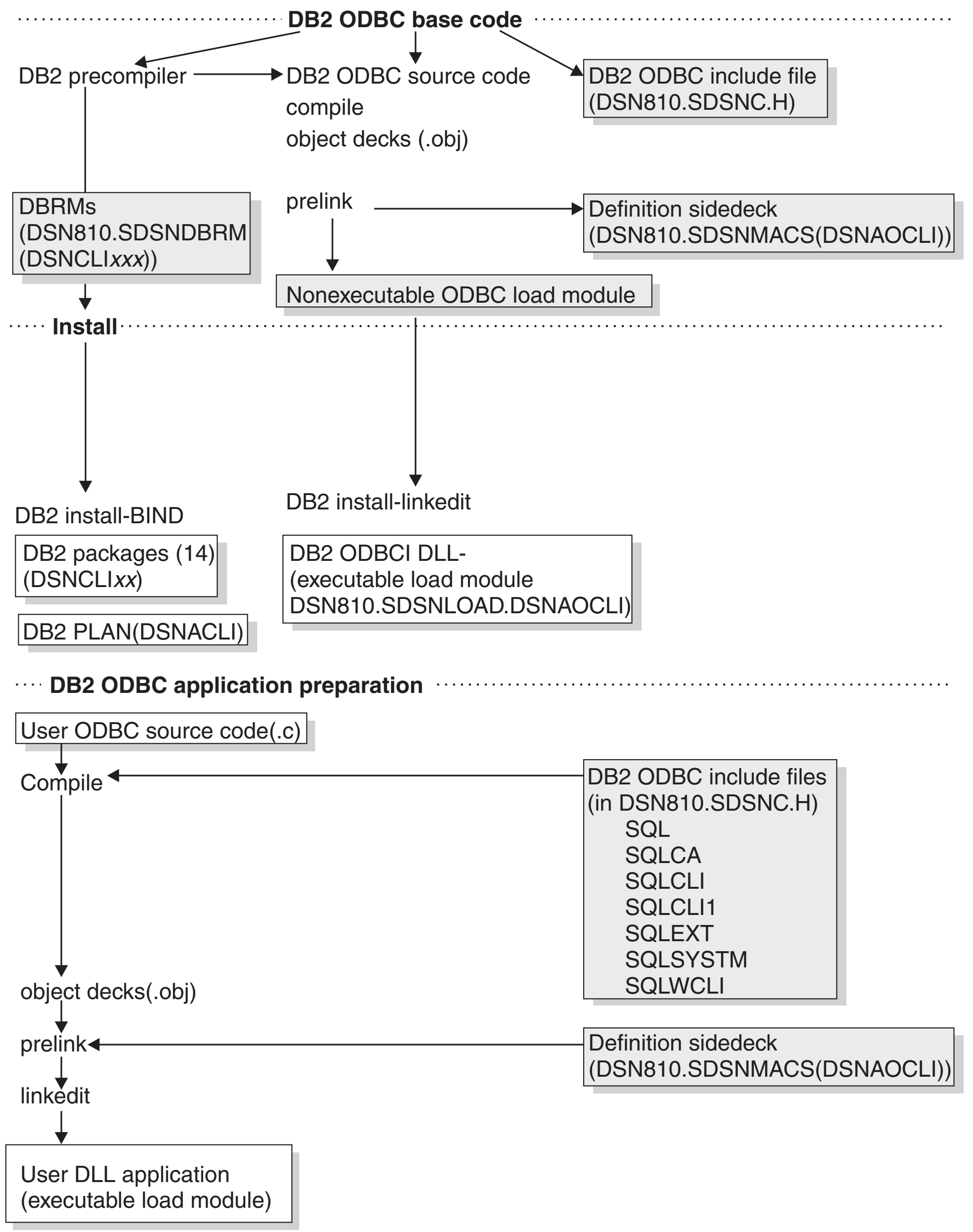


Figure 7. DB2 ODBC driver installation and application preparation

The following sections describe the requirements and steps that are necessary to run a DB2 ODBC application.

- "DB2 ODBC application requirements"
- "Application preparation and execution steps" on page 46

DB2 ODBC application requirements

To successfully build a DLL application, you must ensure that the correct compile, pre-link, and link-edit options are used. In particular, your application must generate the appropriate DLL linkage for the exported DB2 ODBC DLL functions.

The C++ compiler always generates DLL linkage. However, the C compiler only generates DLL linkage if the DLL compile option is used. Failure to generate the necessary DLL linkage can cause the prelinker and linkage editor to issue warning messages for unresolved references to DB2 ODBC functions.

The minimum requirements for a DB2 UDB for z/OS Version 8 ODBC application are as follows:

- z/OS Version 1 Release 3 Application Enablement optional feature for C/C++.

  If the C compiler is used, then the DLL compiler option must be specified.
- z/OS Version 1 Release 3 Language Environment Application Enablement base feature.
- The DB2 ODBC application must be written and link-edited to execute with a 31-bit addressing mode, AMODE(31).

Important: If you build a DB2 ODBC application in z/OS UNIX, you can use the `c89` or `cxx` compile commands to compile your application. Although you compile your application in the z/OS UNIX environment, you can directly reference the non-HFS DB2 ODBC data sets in the `c89` or `cxx` commands. You do not need to copy the DB2 ODBC product files to HFS.

Application preparation and execution steps

The following steps describe application preparation and execution:

- "Step 1. Compile the application"
- "Step 2. Pre-link and link-edit the application" on page 47
- "Step 3. Execute the application" on page 48

DB2 ODBC provides online samples for installation verification:

DSN8O3VP
A sample C application. You can use this sample to verify that your DB2 ODBC 3.0 installation is correct. See "DSN8O3VP sample application" on page 531.

DSN8OIVP
A sample C application. You can use this sample to verify that your DB2 ODBC 2.0 installation is correct.

DSNTEJ8
Sample JCL. You can use this sample to compile, pre-link, link-edit, and execute the sample application DSN8O3VP.

The DSN8O3VP, DSN8OIVP, and DSNTEJ8 online samples are available in DSN810.SDSNSAMP. Using these samples for guidance is highly recommended if you prepare and execute an application.

Using ODBC samples in the z/OS UNIX environment: To use the ODBC samples DSN8O3VP and DSN8OIVP in the z/OS UNIX environment, copy DSN8O3VP or DSN8OIVP from the sample data set to HFS. The following example copies DSN8O3VP to HFS in the directory user/db2, which is considered to be the user's directory:

```
oput  'dsn810.sdsnsamp(dsn8o3vp)' '/usr/db2/dsn8o3vp.c' TEXT
```

Step 1. Compile the application

Include the following directive in the header of your DB2 ODBC application:

```
#include <sqlcli1.h>
```

The sqlcli1.h file includes all information that is required for compiling your DB2 ODBC application. All DB2 ODBC header files, including sqlcli1.h, that define the function prototypes, constants, and data structures that are needed for a DB2 ODBC application are shipped in the DSN810.SDSNC.H data set. Therefore, you must add this data set to your SYSPATH concatenation when you compile your DB2 ODBC application.

For an example of a compile job, use the DSNTEJ8 online sample in DSN810.SDSNSAMP.

To compile an ODBC C application in the z/OS UNIX environment, use the `c89` compile command with the `-W l` specification and specify the `'dll'` option. (The `'dll'` option enables the use of the DB2 ODBC driver for C applications.)

Example: To compile a C application named dsn8o3vp.c that resides in the current working directory, use the following `c89` compile command:

```
c89 -c -W 'c,dll,long,source,list' -
 -I"//'DSN810.SDSNC.H'" \
 dsn8o3vp.c
```

To compile an ODBC C++ application in the z/OS UNIX environment, use the `cxx` compile command with the `-W l` specification. and specify the `'dll'` option. (You must specify the `'dll'` option to enable the use of the DB2 ODBC driver if you use the `cxx` compile command to compile any C parts.)

Example: To compile a C++ application named dsn8o3vp.c that resides in the current working directory, use the following `cxx` compile command:

```
cxx -c -W 'c,long,source,list' -
 -I"//'DSN810.SDSNC.H'" \
 dsn8o3vp.c
```

Step 2. Pre-link and link-edit the application

Before you can link-edit your DB2 ODBC application, you must pre-link your application with a DB2 ODBC definition side-deck that is provided with Version 8 of DB2 UDB for z/OS. To make the DB2 ODBC definition side-decks available in the UNIX environment, see “Setting up the z/OS UNIX environment” on page 44.

For more information about DLL, see *z/OS C/C++ Programming Guide*.

The definition side-deck defines all of the exported functions in the DB2 ODBC dynamic load library, DSNAOCLI. DSNAOCLI resides in the DSN810.SDSNMACS data set. To compile an application, you must include the DSNAOCLI member as input to the prelinker by specifying DSNAOCLI in the pre-link SYSIN data definition statement concatenation. For an example of a z/OS pre-link and link-edit job, use the DSNTEJ8 online sample in DSN810.SDSNSAMP.

If you build a DB2 ODBC application in the z/OS UNIX environment, you can use the `c89` command to pre-link and link-edit your application. You need to include the DB2 ODBC definition side-deck as one of the input data sets to the `c89` or `cxx` command. If you are compiling C code, specify `'dll'` as one of the link-edit options. Before you can use a DB2 ODBC definition side-deck for input to the `c89` or `cxx` command, you must either specify an alias that uses `.EXP` for the last qualifier, or change the value of the _XSUFFIX_HOST z/OS environmental variable.

Example: Assume that you have compiled an application named myapp.c to create a myapp.o file in the current working directory. Assume that you also specified an alias that uses `.EXP` as the last qualifier for the DB2 ODBC definition side-deck. You use the following `c89` command to pre-link and link-edit the C application:

```
c89  -W l,p,map,noer  -W l,dll,AMODE=31,map \
 -o dsn8o3vp dsn8o3vp.o "//'DSN810.SDSNC.EXP(DSNAOCLI)'"
```

You use the following `cxx` command to pre-link and link-edit the C++ application:

```
cxx  -W l,p,map,noer  -W l,dll,AMODE=31,map \
 -o dsn8o3vp dsn8o3vp.o "//'DSN810.SDSNC.EXP(DSNAOCLI)'"
```

Example: Assume that you have compiled an application named myapp.c to create a myapp.o file in the current working directory. Assume that you also changed the value of the _XSUFFIX_HOST environmental variable to SDSNMACS. You use the following `c89` command to pre-link and link-edit the C application:

```
c89 -W l,p,map,noer -W l,dll,AMODE=31,map -o dsn8o3vp dsn8o3vp.o
"//'DSN810.SDSNMACS(DSNAOCLI)'"
```

You use the following `cxx` command to pre-link and link-edit the C++ application:

```
cxx -W l,p,map,noer -W l,dll,AMODE=31,map -o dsn8o3vp dsn8o3vp.o
"//'DSN810.SDSNMACS(DSNAOCLI)'"
```

Step 3. Execute the application

DB2 ODBC applications must access the DSN810.SDSNLOAD data set at execution time. The SDSNLOAD data set contains both the DB2 ODBC dynamic load library and the attachment facility, which is used to communicate with DB2.

In addition, the DB2 ODBC driver accesses the DB2 UDB for z/OS load module DSNHDECP. DSNHDECP contains the coded character set ID (CCSID) information that DB2 UDB for z/OS uses.

A default DSNHDECP is shipped with DB2 UDB for z/OS in the DSN810.SDSNLOAD data set. However, if the values provided in the default DSNHDECP are not appropriate for your site, you can create a new DSNHDECP during installation of DB2 UDB for z/OS. If you create a site-specific DSNHDECP during installation, you concatenate the data set that contains the new DSNHDECP before the DSN810.SDSNLOAD data set in your STEPLIB or JOBLIB data definition statement.

For an example of an execution job, see the DSNTEJ8 online sample in DSN810.SDSNSAMP.

To execute a DB2 ODBC application in the z/OS UNIX environment, you need to include the DSN810.SDSNEXIT and DSN810.SDSNLOAD data sets in the data set concatenation of your STEPLIB environmental variable. The STEPLIB environmental variable can be set in your .profile file with the following statement:

```
export STEPLIB=DSN810.SDSNEXIT:DSN810.SDSNLOAD
```

Defining a subsystem

You can define a DB2 subsystem to DB2 ODBC in two different ways. You can identify the DB2 subsystem by specifying the MVSDEFAULTSSID keyword in the common section of initialization file. If the MVSDEFAULTSSID keyword does not exist in the initialization file, DB2 ODBC uses the default subsystem name specified in the DSNHDECP load module that was created when DB2 was installed. Therefore, you should ensure that DB2 ODBC can find the intended DSNHDECP when your application issues the SQLAllocHandle() call (with *HandleType* set to SQL_HANDLE_ENV).

The DSNHDECP load module is usually link-edited into the DSN810.SDSNEXIT data set. In this case, your STEPLIB data definition statement includes:

```
//STEPLIB    DD  DSN=DSN810.SDSNEXIT,DISP=SHR
//           DD  DSN=DSN810.SDSNLOAD,DISP=SHR
...
```

DB2 ODBC initialization file

A set of optional keywords can be specified in a DB2 ODBC *initialization file*, an EBCDIC file that stores default values for various DB2 ODBC configuration options. Because the initialization file has EBCDIC text, it can be updated using a file editor, such as the TSO editor.

For most applications, use of the DB2 ODBC initialization file is not necessary. However, to make better use of IBM RDBMS features, the keywords can be specified to:
- Help improve the performance or usability of an application.
- Provide support for applications written for a previous version of DB2 ODBC.
- Provide specific workarounds for existing ODBC applications.

The following sections describe how to create the initialization file and define the keywords:
- "Using the initialization file"
- "Initialization keywords" on page 51

Using the initialization file

The DB2 ODBC initialization file is read at application run time. The file can be specified by either a DSNAOINI data definition statement or by defining a DSNAOINI z/OS UNIX environmental variable. DB2 ODBC opens the DSNAOINI data set allocated in your JCL first. If a DSNAOINI data set is not allocated, then DB2 ODBC opens the environmental variable data set.

The initialization file specified can be either a traditional z/OS data set or an HFS file under the z/OS UNIX environment. For z/OS data sets, the record format of the initialization file can be either fixed or variable length.

The following examples use a DSNAOINI JCL data definition statement to specify the DB2 ODBC initialization file types supported:

Sequential data set USER1.DB2ODBC.ODBCINI:

```
//DSNAOINI DD DSN=USER1.DB2ODBC.ODBCINI,DISP=SHR
```

Partitioned data set USER1.DB2ODBC.DATA, member ODBCINI:

```
//DSNAOINI DD DSN=USER1.DB2ODBC.DATA(ODBCINI),DISP=SHR
```

Inline JCL DSNAOINI DD specification:

```
//DSNAOINI DD *
  [COMMON]
  MVSDEFAULTSSID=V61A
/*
```

HFS file /u/user1/db2odbc/odbcini:

```
//DSNAOINI DD PATH='/u/user1/db2odbc/odbcini'
```

The following examples of z/OS UNIX export statements define the DB2 ODBC DSNAOINI z/OS UNIX environmental variable for the DB2 ODBC initialization file types supported:

HFS fully qualified file /u/user1/db2odbc/odbcini:

```
export DSNAOINI="/u/user1/db2odbc/odbcini"
```

HFS file ./db2odbc/odbcini, relative to the present working directory of the application:

```
export DSNAOINI="./db2odbc/odbcini"
```

Sequential data set USER1.ODBCINI:

```
export DSNAOINI="USER1.ODBCINI"
```

Redirecting to use a file that is specified by another previously allocated DD statement, MYDD:

```
export DSNAOINI="//DD:MYDD"
```

Partitioned data set USER1.DB2ODBC.DATA, member ODBCINI:

```
export DSNAOINI="USER1.DB2ODBC.DATA(ODBCINI)"
```

When specifying an HFS file, the value of the DSNAOINI environmental variable must begin with either a single forward slash (/), or a period followed by a single forward slash (./). If a setting starts with any other characters, DB2 ODBC assumes that a z/OS data set name is specified.

Allocation precedence: DB2 ODBC opens the DSNAOINI data set allocated in your JCL first. If a DSNAOINI data set is not allocated, then DB2 ODBC opens the environmental variable data set.

Initialization file structure

The initialization file consists of the following three sections, or stanzas:

Common section
Contains parameters that are global to all applications using this initialization file.

Subsystem section
Contains parameter values unique to that subsystem.

Data source sections
Contain parameter values to be used only when connected to that data source. You can specify zero or more data source sections.

Each section is identified by a syntactic identifier enclosed in square brackets. Specific guidelines for coding square brackets are described in the list item below marked ’Attention’.

The syntactic identifier is either the literal 'common', the subsystem ID or the data source (location name). For example:

[*data-source-name***]**

This is 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 data source must be located below the data source section header.

- The keyword settings in each section apply only to the data source name 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, see "Initialization keywords."
- If a data source name is not found in the DB2 ODBC initialization file, the default values for these keywords are in effect.
- Comment lines are introduced by having a semicolon in the first position of a new line.
- Blank lines are also permitted. If duplicate entries for a keyword exist, the first entry is used (and no warning is given).

Important: You can avoid common errors by ensuring that the following contents of the initialization file are accurate:
- Square brackets: The square brackets in the initialization file must consist of the correct EBCDIC characters. The open square bracket must use the hexadecimal characters X'AD'. The close square bracket must use the hexadecimal characters X'BD'. DB2 ODBC does not recognize brackets if coded differently.
- Sequence numbers: The initialization file cannot accept sequence numbers. All sequence numbers must be removed.

The following sample is a DB2 ODBC initialization file with a common stanza, a subsystem stanza, and two data source stanzas.

```
; This is a comment line...
; Example COMMON stanza
[COMMON]
MVSDEFAULTSSID=V81A

; Example SUBSYSTEM stanza for V81A subsystem
[V81A]
MVSATTACHTYPE=CAF
PLANNAME=DSNACLI

; Example DATA SOURCE stanza for STLEC1 data source
[STLEC1]
AUTOCOMMIT=0
CONNECTTYPE=2

; Example DATA SOURCE stanza for STLEC1B data source
[STLEC1B]
CONNECTTYPE=2
CURSORHOLD=0
```

Initialization keywords

The initialization keywords are described in this section. The section (common, subsystem, or data source) in the initialization file where each keyword must be defined is identified.

APPLTRACE = 0 | 1

This keyword is placed in the common section.

The APPLTRACE keyword controls whether the DB2 ODBC application trace is enabled. The application trace is designed for diagnosis of application errors. If enabled, every call to any DB2 ODBC API from the application is traced, including input parameters. The trace is written to the file specified on the APPLTRACEFILENAME keyword.

0: Disabled (default)
1: Enabled

For more information about using the APPLTRACE keyword, see “Application trace” on page 477.

Important: This keyword was renamed in Version 7. DB2 ignores the Version 6 keyword CLITRACE.

APPLTRACEFILENAME = *dataset_name*
This keyword is placed in the common section.

APPLTRACEFILENAME is only used if a trace is started by the APPLTRACE keyword. When APPLTRACE is set to 1, use the APPLTRACEFILENAME keyword to identify a z/OS data set name or z/OS UNIX environment HFS file name that records the DB2 ODBC application trace. “Diagnostic trace” on page 479 provides detailed information about specifying file name formats.

Important: This keyword was renamed. DB2 ignores the Version 6 keyword TRACEFILENAME.

AUTOCOMMIT = 1 | 0
This keyword is placed in the data source section.

To be consistent with ODBC, DB2 ODBC defaults with AUTOCOMMIT on, which means each statement is treated as a single, complete transaction. This keyword can provide an alternative default, but is only used if the application does not specify a value for AUTOCOMMIT as part of the program.
1: on (default)
0: off

Most ODBC applications assume the default of AUTOCOMMIT is on. Extreme care must be used when overriding this default during run time as the application might depend on this default to operate properly.

Although you can specify only two different values for this keyword, you can also specify whether AUTOCOMMIT is enabled in a distributed unit of work (DUW) environment. If a connection is part of a coordinated DUW, and AUTOCOMMIT is not set, the default does not apply; implicit commits arising from autocommit processing are suppressed. If AUTOCOMMIT is set to 1, and the connection is part of a coordinated DUW, the implicit commits are processed. This can result in severe performance degradations, and possibly other unexpected results elsewhere in the DUW system. However, some applications might not work at all unless this is enabled.

A thorough understanding of the transaction processing of an application is necessary, especially applications written by a third party, before applying it to a DUW environment.

To enable global transaction processing in an application, specify AUTOCOMMIT=0, MULTICONTEXT=0, and MVSATTACHTYPE=RRSAF. See “Using global transactions” on page 405 for more information.

BITDATA = 1 | 0
This keyword is placed in the data source section.

The BITDATA keyword allows you to specify whether ODBC binary data types, SQL_BINARY, SQL_VARBINARY, and 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.

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. This is the default.

0: Disabled.

CLISCHEMA = *schema_name*

This keyword is placed in the data source section.

The CLISCHEMA keyword lets you indicate the schema of the DB2 ODBC shadow catalog tables or views to search when you issue an ODBC catalog function call. The character string that you use for *schema_name* must not exceed 128 bytes. For example, if you specify CLISCHEMA=PAYROLL, the ODBC catalog functions that normally reference DB2 catalog tables (SYSIBM schema), will reference the following views of the DB2 ODBC shadow catalog tables:

- PAYROLL.COLUMNS
- PAYROLL.TABLES
- PAYROLL.COLUMNPRIVILEGES
- PAYROLL.TABLEPRIVILEGES
- PAYROLL.SPECIALCOLUMNS
- PAYROLL.PRIMARYKEYS
- PAYROLL.FOREIGNKEYS
- PAYROLL.TSTATISTICS
- PAYROLL.PROCEDURES

You must build the DB2 ODBC shadow catalog tables and optional views before using the CLISCHEMA keyword. If this keyword is not specified, the ODBC catalog query APIs reference the DB2 (SYSIBM) system tables by default.

COLLECTIONID = *collection_id*

This keyword is placed in the data source section.

The COLLECTIONID keyword allows you to specify the collection identifier that is used to resolve the name of the package allocated at the server. This package supports the execution of subsequent SQL statements.

The value is a character string and must not exceed 128 characters. It can be overridden by executing the SET CURRENT PACKAGESET statement.

CONNECTTYPE = 1 | 2

This keyword is placed in the common section.

The CONNECTTYPE keyword allows you to specify the default connection type for all connections to data sources.

1: Multiple concurrent connections, each with its own commit scope. If MULTICONTEXT=0 is specified, a new connection might not be added unless the current transaction on the current connection is on a transaction boundary (either committed or rolled back). This is the default.

2: Coordinated connections where multiple data sources participate under the same distributed unit of work. CONNECTTYPE=2 is ignored if MULTICONTEXT=1 is specified.

CURRENTAPPENSCH = EBCDIC | UNICODE | ASCII

This keyword is placed in the common section.

Use the CURRENTAPPENSCH keyword to specify which encoding scheme (UNICODE, EBCDIC, or ASCII) the ODBC driver uses for input and output host variable data, SQL statements, and input and output ODBC API character string arguments. When this keyword is specified in the initialization file, a SET CURRENT APPLICATION ENCODING SCHEME statement is sent to the data source following a successful connect. If this keyword is not present, then the driver will assume EBCDIC as the default application encoding scheme.

CURRENTFUNCTIONPATH = *″'schema1', 'schema2' ,...″*

This keyword is placed in the data source section.

Use the CURRENTFUNCTIONPATH keyword to define the path that resolves unqualified user-defined functions, distinct types, and stored procedure references that are used in dynamic SQL statements. It contains a list of one or more schema names, which are used to set the CURRENT PATH special register using the SET CURRENT PATH SQL statement upon connection to the data source. Each schema name in the keyword string must be delimited with single quotes and separated by commas. The entire keyword string must be enclosed in double quotes and must not exceed 2048 characters.

The default value of the CURRENT PATH special register is:

```
"SYSIBM", "SYSFUN", "SYSPROC", X
```

X is the value of the USER special register as a delimited identifier. The schemas SYSIBM, SYSFUN, and SYSPROC do not need to be specified. If any of these schemas is not included in the current path, DB2 implicitly assumes each schema name begins the path, in the order shown above. The order of the schema names in the path determines the order in which the names are resolved. For more detailed information about schema name resolution, see *DB2 SQL Reference*.

Unqualified user-defined functions, distinct types, and stored procedures are searched from the list of schemas specified in the CURRENTFUNCTIONPATH setting in the order specified. If the user-defined function, distinct type, or stored procedures is not found in a specified schema, the search continues in the schema specified next in the list. For example:

```
CURRENTFUNCTIONPATH="'USER01', 'PAYROLL', 'SYSIBM', 'SYSFUN', 'SYSPROC'"
```

This example of CURRENTFUNCTIONPATH settings searches schema ″USER01″, followed by schema ″PAYROLL″, followed by schema ″SYSIBM″, and so on.

Although the SQL statement CALL is a static statement, the CURRENTFUNCTIONPATH setting affects a CALL statement if the stored procedure name is specified with a host variable (making the CALL statement a pseudo-dynamic SQL statement). This is always the case for a CALL statement processed by DB2 ODBC.

CURRENTSQLID = *current_sqlid*

This keyword is placed in the data source section.

The CURRENTSQLID keyword is valid only for those DB2 DBMSs that support SET CURRENT SQLID (such as DB2 UDB for z/OS). If this keyword is present, then a SET CURRENT SQLID statement is sent to the DBMS after a successful connect. This allows users and the application to name SQL objects without having to qualify by schema name. The value that you specify for *current_sqlid* must be no more than 128 bytes.

Do not specify this keyword if you are binding the DB2 ODBC packages with DYNAMICRULES(BIND).

CURSORHOLD = 1 | 0

This keyword is placed in the data source section.

The CURSORHOLD keyword controls the effect of a transaction completion on open cursors.

1: Cursor hold. The cursors are not destroyed when the transaction is committed. This is the default.
0: Cursor no hold. The cursors are destroyed when the transaction is committed.

Cursors are always destroyed when transactions are rolled back.

Specify zero for this keyword to improve application performance when both the following conditions are true:

- The application does not behave dependently on SQL_CURSOR_COMMIT_BEHAVIOR or SQL_CURSOR_ROLLBACK_BEHAVIOR information that SQLGetInfo() returns.
- The application does not require cursors to be preserved from one transaction to the next.

The DBMS operates more efficiently as resources no longer need to be maintained after the end of a transaction.

DBNAME = *dbname*

This keyword is placed in the data source section.

The DBNAME keyword is only used when connecting to DB2 UDB for z/OS, and only if (*base*) table catalog information is requested by the application.

If a large number of tables exist in the DB2 UDB for z/OS subsystem, a *dbname* can be specified to reduce the time it takes for the database to process the catalog query for table information, and reduce the number of tables returned to the application.

The value of the *dbname* keyword maps to the DBNAME column in the DB2 UDB for z/OS catalog tables. If no value is specified, or if views, synonyms, system tables, or aliases are also specified using TABLETYPE, only table information is restricted; views, aliases, and synonyms are not restricted with DBNAME. This keyword can be used in conjunction with SCHEMALIST and TABLETYPE to further limit the number of tables for which information is returned.

DIAGTRACE = 0 | 1

This keyword is placed in the common section.

The DIAGTRACE keyword lets you enable the DB2 ODBC diagnostic trace.

0: The DB2 ODBC diagnostic trace is not enabled. No diagnostic data is captured. This is the default.

You can enable the diagnostic trace using the DSNAOTRC command when the DIAGTRACE keyword is set to 0.
1: The DB2 ODBC diagnostic trace is enabled. Diagnostic data is recorded in the application address space. If you include a DSNAOTRC data definition statement in your job or TSO logon procedure that identifies a z/OS data set or a z/OS UNIX environment HFS file name, the trace is externalized at normal program termination. You can format the trace by using the DSNAOTRC trace formatting command.

For more information about using the DIAGTRACE keyword and the DSNAOTRC command, see “Diagnostic trace” on page 479.

Important: This keyword was renamed. DB2 ignores the Version 6 keyword TRACE.

DIAGTRACE_BUFFER_SIZE = *buffer size*
This keyword is placed in the common section.

The DIAGTRACE_BUFFER_SIZE keyword controls the size of the DB2 ODBC diagnostic trace buffer. This keyword is only used if a trace is started by using the DIAGTRACE keyword.

buffer size is an integer value that represents the number of bytes to allocate for the trace buffer. The buffer size is rounded down to a multiple of 65536 (64K). If the value specified is less than 65536, then 65536 is used. The default value for the trace buffer size is 65536.

If a trace is active, this keyword is ignored.

Important: DB2 ignores the Version 6 keyword TRACE_BUFFER_SIZE.

DIAGTRACE_NO_WRAP = 0 | 1
This keyword is placed in the common section.

The DIAGTRACE_NO_WRAP keyword controls the behavior of the DB2 ODBC diagnostic trace when the DB2 ODBC diagnostic trace buffer fills up. This keyword is only used if a trace is started by the DIAGTRACE keyword.
0: The trace table is a wraparound trace. In this case, the trace remains active to capture the most current trace records. This is the default.
1: The trace stops capturing records when the trace buffer fills. The trace captures the initial trace records that were written.

If a trace is active, this keyword is ignored.

Important: This keyword was renamed. DB2 ignores the Version 6 keyword name TRACE_NO_WRAP.

GRAPHIC =0 | 1 | 2 | 3
This keyword is placed in the data source section.

The GRAPHIC keyword controls whether DB2 ODBC reports 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 data source for the current connection. These are not native ODBC types but have been added to expose these types to an application connected to a DB2 family product.

0: disabled (default)

1: enabled

2: report the length of graphic columns returned by DESCRIBE in number of bytes rather than DBCS characters. This applies to all DB2 ODBC and ODBC functions that return length or precision either on the output argument or as part of the result set.

3: settings 1 and 2 combined; that is, GRAPHIC=3 achieves the combined effect of 1 and 2.

The default is that GRAPHIC is not returned because many applications do not recognize this data type and cannot provide proper handling.

MAXCONN = 0 | *positive number*
This keyword is placed in the common section.

The MAXCONN keyword is used to specify the maximum number of connections allowed for each DB2 ODBC application program. This can be used by an administrator as a governor for the maximum number of connections established by each application.

0: can be used to represent *no limit*; that is, an application is allowed to open up as many connections as permitted by the system resources. This is the default.

positive number: set the keyword to any positive number to specify the maximum number of connections each application can open.

This parameter limits the number of SQLConnect() statements that the application can successfully issue. In addition, if the application is executing with CONNECT (type 1) semantics, then this value specifies the number of logical connections. Only one physical connection to either the local DB2 subsystem or a remote DB2 subsystem or remote DRDA-1 or DRDA-2 server is made at one time.

MULTICONTEXT = 0 | 1

This keyword is placed in the common section.

The MULTICONTEXT keyword controls whether each connection in an application can be treated as a separate unit of work with its own commit scope that is independent of other connections.

0: The DB2 ODBC code does not create an independent *context* for a data source connection. Connection switching among multiple data sources governed by the CONNECTTYPE=1 rules is not allowed unless the current transaction on the current connection is on a transaction boundary (either committed or rolled back). This is the default.

Specify MULTICONTEXT=0 and MVSATTACHTYPE=RRSAF to allow an ODBC application to create and manage its own contexts using the z/OS Context Services. With these services, an application can manage its own contexts outside of ODBC with each context operating as an independent unit of work.

DB2 ODBC support for external contexts is disabled if the application is running as DB2 ODBC stored procedure. See “External contexts” on page 440 for more information.

To enable global transaction processing in an application, specify AUTOCOMMIT=0, MULTICONTEXT=0, and MVSATTACHTYPE=RRSAF. See “Using global transactions” on page 405 for more information.

1: The DB2 ODBC code creates an independent context for a data source connection at the connection handle level when SQLAllocHandle() is issued. Each connection to multiple data sources is governed by CONNECTTYPE=1 rules and is associated with an independent DB2 thread. Connection switching among multiple data sources is not prevented due to the commit status of the transaction; an application can use multiple connection handles without having to perform a commit or rollback on a connection before switching to another connection handle.

The application can use SQLGetInfo() with *InfoType* set to SQL_MULTIPLE_ACTIVE_TXN to determine whether MULTICONTEXT=1 is supported.

MULTICONTEXT=1 is ignored if any of these conditions are true:

- The application created a DB2 thread before invoking DB2 ODBC. This is always the case for a stored procedure using DB2 ODBC.

- The application created and switched to a private context using z/OS Context Services before invoking DB2 ODBC.
- The application started a unit of recovery with any RRS resource manager (for example, IMS) before invoking DB2 ODBC.
- MVSATTACHTYPE=CAF is specified in the initialization file.
- The operating system level does not support Unauthorized Context Services.

See "DB2 ODBC support of multiple contexts" on page 435 for more details.

MVSATTACHTYPE = CAF | RRSAF
This keyword is placed in the subsystem section.

The MVSATTACHTYPE keyword is used to specify the DB2 UDB for z/OS attachment type that DB2 ODBC uses to connect to the DB2 UDB for z/OS address space. This parameter is ignored if the DB2 ODBC application is running as a DB2 UDB for z/OS stored procedure. In that case, DB2 ODBC uses the attachment type that was defined for the stored procedure.

CAF: DB2 ODBC uses the DB2 UDB for z/OS call attachment facility (CAF). This is the default.

RRSAF: DB2 ODBC uses the DB2 UDB for z/OS Resource Recovery Services attachment facility (RRSAF).

Specify MVSATTACHTYPE=RRSAF and MULTICONTEXT=0 to allow an ODBC application to create and manage its own contexts using the z/OS Context Services. See MULTICONTEXT=0 for more information.

For transactions on a global connection, specify AUTOCOMMIT=0, MULTICONTEXT=0, and MVSATTACHTYPE=RRSAF to complete global transaction processing.

To enable global transaction processing in an application, specify MVSATTACHTYPE=RRSAF, AUTOCOMMIT=0, and MULTICONTEXT=0. See "Using global transactions" on page 405 for more information.

MVSDEFAULTSSID = *ssid*
This keyword is placed in the common section.

The MVSDEFAULTSSID keyword specifies the default DB2 subsystem to which the application is connected when invoking the SQLAllocHandle() function (with *HandleType* set to SQL_HANDLE_ENV). Specify the DB2 subsystem name or group attachment name (if used in a data sharing group) to which connections will be made. The default subsystem is 'DSN'.

OPTIMIZEFORNROWS = *integer*
This keyword is placed in the data source section.

The OPTIMIZEFORNROWS keyword appends the ″OPTIMIZE FOR n ROWS″ clause to every select statement, where n is an integer larger than 0. The default action is not to append this clause.

For more information about the effect of the OPTIMIZE FOR n ROWS clause, see *DB2 SQL Reference*.

PATCH2 = *patch number*
This keyword is placed in the data source section.

The PATCH2 keyword specifies a workaround for known problems with ODBC applications. To set multiple PATCH2 values, list the values sequentially, separated by commas. For example, if you want patches 300, 301, and 302, specify PATCH2= "300,301,302" in the initialization file. The valid values for the PATCH2 keyword are:

0: No workaround (default).
300

PATCH2=300 behavior: SQLExecute() and SQLExecDirect() will return SQL_NO_DATA_FOUND instead of SQL_SUCCESS when SQLCODE=100. In this case, a delete or update affected no rows, or the result of the subselect of an insert statement is empty.

Table 10 explains how PATCH2 settings affect return codes.

Table 10. PATCH2 settings and SQL return codes

SQL statement	SQLExecute() and SQLExecDirect() return value
A searched update or searched delete and no rows satisfy the search condition	• SQL_SUCCESS without a patch (PATCH2=0) • SQL_NO_DATA_FOUND with a patch (PATCH2=300)
A mass delete or update and no rows satisfy the search condition	• SQL_SUCCESS_WITH_INFO without a patch (PATCH2=0) • SQL_NO_DATA_FOUND with a patch (PATCH2=300)
A mass delete or update and one or more rows satisfy the search condition	SQL_SUCCESS_WITH_INFO without a patch (PATCH2=0) or with a patch (PATCH2=300)

In ODBC 3.0, applications do not need to set the patch on. ODBC 3.0 behavior is equivalent to setting PATCH2=300.

PLANNAME = *planname*
This keyword is placed in the subsystem section.

The PLANNAME keyword specifies the name of the DB2 UDB for z/OS PLAN that was created during installation. A PLAN name is required when initializing the application connection to the DB2 UDB for z/OS subsystem which occurs during the processing of the SQLAllocHandle() call (with *HandleType* set to SQL_HANDLE_ENV).

If no PLANNAME is specified, the default value DSNACLI is used.

SCHEMALIST = *"***'***schema1***'**, **'***schema2***'** **,...***"*
This keyword is placed in the data source section.

The SCHEMALIST keyword specifies a list of schemas in the data source. If a database contains a large number of tables, you can specify a schema list to reduce the time it takes for the application to query table information and 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 UDB for z/OS, 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 keyword string is 2048 bytes.

This keyword can be used in conjunction with DBNAME and TABLETYPE to further limit the number of tables for which information is returned.

SCHEMALIST is used to provide a more restrictive default in the case of those applications that always give a list of every table in the DBMS. This improves performance of the table list retrieval in cases where the user is only interested in seeing the tables in a few schemas.

SYSSCHEMA = *sysschema*

This keyword is placed in the data source section. This keyword is placed in the data source section. The value that you specify for *sysschema* must be no longer than 128 bytes.

The SYSSCHEMA keyword indicates an alternative schema to be searched in place of the SYSIBM (or SYSTEM, QSYS2) schemas when the DB2 ODBC and ODBC catalog function calls are issued to obtain 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 DB2 catalog tables:

- SYSCOLAUTH
- SYSCOLUMNS
- SYSDATABASE
- SYSFOREIGNKEYS
- SYSINDEXES
- SYSKEYS
- SYSPARMS
- SYSRELS
- SYSROUTINES
- SYSSYNONYMS
- SYSTABAUTH
- SYSTABLES

For example, if the set of views for the catalog tables are in the ACME schema, the view for SYSIBM.SYSTABLES is ACME.SYSTABLES, and SYSSCHEMA should then be set to ACME.

Defining and using limited views of the 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:

- SYSIBM on DB2 UDB for z/OS
- SYSTEM on DB2 for VSE & VM
- QSYS2 on DB2 UDB for iSeries

This keyword can be used in conjunction with SCHEMALIST, TABLETYPE (and DBNAME on DB2 UDB for z/OS) to further limit the number of tables for which information is returned.

TABLETYPE=″'TABLE' | ,'ALIAS' | ,'VIEW' | , 'SYSTEM TABLE' | ,'SYNONYM'″

This keyword is placed in the data source section.

The TABLETYPE keyword specifies a list of one or more table types. If a large number of tables are defined in the data source, you can specify a table type string to reduce the time it takes for the application to query table information and the number of tables the application lists.

Any number of the values can be specified, but each type must be delimited with single quotes, separated by commas, and in upper case. The entire string must also be enclosed in double quotes, for example:

```
TABLETYPE="'TABLE','VIEW'"
```

This keyword can be used in conjunction with DBNAME and SCHEMALIST to further limit the number of tables for which information is returned.

| TABLETYPE is used to provide a default for the SQLTables() call, which retrieves a list of tables, views, aliases, and synonyms in a data source. 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 *szTableType* argument on the function call, that argument value overrides 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 UDB for z/OS subsystem.

THREADSAFE= 1 | 0
This keyword is placed in the common section.

The THREADSAFE keyword controls whether DB2 ODBC uses *POSIX mutexes* to make the DB2 ODBC code *threadsafe* for multiple concurrent or parallel Language Environment threads.

1: The DB2 ODBC code is threadsafe if the application is executing in a POSIX(ON) environment. Multiple Language Environment threads in the process can use DB2 ODBC. The threadsafe capability cannot be provided in a POSIX(OFF) environment. This is the default.

0: The DB2 ODBC code is not threadsafe. This reduces the overhead of serialization code in DB2 ODBC for applications that are not *multithreaded*, but provides no protection for concurrent Language Environment threads in applications that are multithreaded.

TXNISOLATION = 1 | 2 | 4 | 8 | 32
This keyword is placed in the data source section.

The TXNISOLATION keyword 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, DB2 UDB for iSeries only)

The words in round brackets are the DB2 equivalents for SQL92 isolation levels. Note that ″no commit″ is not an SQL92 isolation level and is supported only on DB2 UDB for iSeries. See *DB2 Application Programming and SQL Guide* for more information about isolation levels.

UNDERSCORE = 1 | 0
This keyword is placed in the data source section.

The UNDERSCORE keyword specifies 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 parameter only affects catalog function calls that accept search pattern strings. You can set the UNDERSCORE keyword to the following settings:

1: The underscore character (_) acts as a wildcard (default). The underscore is treated as a wildcard matching any one character or none. For example, two tables are defined as follows:

```
CREATE TABLE "OWNER"."KEY_WORDS" (COL1 INT)
CREATE TABLE "OWNER"."KEYWORDS" (COL1 INT)
```

In the example above, SQLTables() (the DB2 ODBC catalog function call that returns table information) returns both the ″KEY_WORDS″ and ″KEYWORDS″ entries if ″KEY_WORDS″ is specified in the table name search pattern argument.

0: The underscore character (_) acts as itself. The underscore is treated literally as an underscore character. If two tables are defined as shown in the example above, SQLTables() returns 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 data source contain underscores.

DB2 ODBC migration considerations

When you migrate from the DB2 UDB for z/OS Version 7 ODBC driver to the DB2 UDB for z/OS Version 8 ODBC driver, you must set up the DB2 ODBC run-time environment as "Setting up the DB2 ODBC run-time environment" on page 41 describes. You must bind the Version 8 DB2 ODBC DBRMs to each data source you want to migrate to the DB2 UDB for z/OS Version 8 ODBC driver. The following steps summarize the migration process:

1. Bind the DBRMs in DSN810.SDSNDBRM to all data sources to which your ODBC applications connect. Unlike DB2 UDB for z/OS Version 7 ODBC, you must specify ENCODING(EBCDIC) when you bind Version 8 ODBC DBRMs to the local DB2 UDB for z/OS subsystem. "Bind DBRMs to create packages" on page 41 list the specific DBRMs you must bind to each type of DB2 server.
2. Create at least one DB2 plan. Use the PKLIST keyword to specify all the packages that you create from the DBRMs that "Bind DBRMs to create packages" on page 41 lists.

The online bind sample is available in DSN810.SDSNSAMP(DSNTIJCL). It is strongly recommended that you use this bind sample as a guide for binding DBRMs to packages and binding an application plan.

"Setting up the z/OS UNIX environment" on page 44 describes special considerations for the z/OS UNIX environment.

Chapter 4. Functions

This chapter provides a description of each DB2 ODBC function. Each of these descriptions include the following sections.
- Purpose
- Syntax
- Function arguments
- Usage
- Return codes
- Diagnostics
- Restrictions
- References

Each section of the function descriptions is described below. DB2 ODBC deprecated functions (which are functions that new DB2 ODBC functions replace) include only the purpose, syntax, and function arguments sections. For more information about deprecated functions, see Appendix E, “Deprecated functions,” on page 525.

Purpose

This section provides a brief overview of the function. It also indicates any additional functions that you should call before or after you use the function that the section describes.[1]

This section also contains a table that indicates the specifications and standards to which the function conforms. The first column indicates whether the function is included in the ODBC specification and identifies the first ODBC version (1.0, 2.0, or 3.0) that includes the specification for the function. The second column indicates whether the function is included in the X/Open CLI CAE specification, and the third column indicates if the function is included in the ISO CLI standard. Table 11 is an example of the specifications table for an ODBC 3.0 function that is included in both the X/Open CLI CAE specification and the ISO CLI standard.

Table 11. Sample function specification table

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

Some functions use a set of attributes that do not apply to all specifications or standards. The restrictions section identifies any significant differences between these specifications and the DB2 ODBC implementation.

Syntax

This section contains the generic C language prototype for the function. If the function is defined by ODBC 3.0, the prototype is identical to that specified in *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*.

All function arguments that are pointers are defined using the FAR macro. This macro is defined out (set to a blank). This is consistent with the ODBC specification.

Function arguments

This section lists each function argument, along with its data type, a description and a indication of whether it is an input or output argument.

1. For functions that are deprecated in DB2 ODBC, the purpose section directs you to the current DB2 ODBC function.

Only SQLGetInfo() and SQLBindParameter() use parameters for both input and output.

Some functions use input or output arguments that are known as *deferred* or *bound* arguments. These arguments are pointers to buffers that you allocate in your application and associate with (or bind to) either a parameter in an SQL statement or a column in a result set. DB2 ODBC accesses these buffers when you execute the SQL statement or retrieve the result set to which the deferred arguments are bound.

Important: For input arguments, ensure that deferred data areas contain valid data when you execute a statement that requires these values. For output arguments, ensure that deferred data areas remain allocated until you finish retrieving results.

Usage
This section provides information about how to use the function and any special considerations. Possible error conditions are not discussed here; this information is listed in the diagnostics section.

Return codes
This section lists all the possible function return codes. When SQL_ERROR or SQL_SUCCESS_WITH_INFO is returned, you can obtain error information by calling SQLGetDiagRec().[2]

See "Diagnostics" on page 22 for more information about return codes.

Diagnostics
This section contains a table that lists the SQLSTATEs that are explicitly returned by DB2 ODBC and indicates the cause of the error. (DB2 ODBC can also return SQLSTATEs that the DBMS generates.) To obtain these SQLSTATEs, call SQLGetDiagRec() on a function that returns SQL_ERROR or SQL_SUCCESS_WITH_INFO.[2]

See "Diagnostics" on page 22 for more information about diagnostics.

Restrictions
This section indicates any differences or limitations between DB2 ODBC and ODBC that can affect an application.[2]

Example
This section contains a code fragment that demonstrates the use of the function, using the generic data type definitions.[2]

See Chapter 3, "Configuring DB2 ODBC and running sample applications," on page 39 for more information about setting up the DB2 ODBC environment and accessing the sample applications.

Related functions
This section lists DB2 ODBC functions that are related to calls to the function that is described.[2]

Function overview

Table 12 provides a complete list of functions, which are divided by task, that DB2 ODBC and Microsoft ODBC 3.0 support. For each function, the table indicates the ODBC 3.0 conformance level, DB2 ODBC support, and a brief description.

The ODBC 3.0 column contains the following values:

2. This section does not appear in the descriptions of functions that are deprecated in DB2 ODBC.

No	Indicates that the function is not supported by ODBC 3.0.
Depr	Indicates that the function is supported but deprecated in ODBC 3.0.
Core	Indicates that the function is part of the ODBC 3.0 Core conformance level.
Lvl 1	Indicates that the function is part of the ODBC 3.0 Level 1 conformance level.
Lvl 2	Indicates that the function is part of the ODBC 3.0 Level 2 conformance level.

The DB2 ODBC support column contains the following values:

No	Indicates that the function is not supported by DB2 ODBC.
Depr	Indicates that the function is supported but deprecated in DB2 ODBC.
Current	Indicates that the function is current for DB2 ODBC. A current function is supported by DB2 ODBC and is not deprecated by another DB2 ODBC function.

A function that is deprecated in ODBC 3.0 is not necessarily deprecated in DB2 ODBC. See Appendix E, “Deprecated functions,” on page 525 for more information about deprecated functions.

Table 12. Function list by category

Function name divided by task	ODBC 3.0	DB2 ODBC	Purpose
Connecting to a data source			
SQLAllocConnect()	Depr	Depr	Obtains a connection handle.
SQLAllocEnv()	Depr	Depr	Obtains an environment handle. One environment handle is used for one or more connections.
SQLAllocHandle()	Core	Current	Obtains a handle.
| SQLBrowseConnect()	Lvl 1	No	Returns successive levels of connection attributes and valid attribute values. When a value is specified for each connection attribute, this function connects to the data source.
SQLConnect()	Core	Current	Connects a to specific driver by data source name, user ID, and password.
| SQLDriverConnect()	Core	Current	Connects to a specific driver with a connection string. **IBM specific:** This function is also extended with the additional IBM keywords that are supported in the ODBC.INI file in the DB2 UDB CLI environment. Within the DB2 UDB for z/OS ODBC environment, the ODBC.INI file has no equivalent.
SQLSetConnection()	No	Current	Connects to a specific data source by connection string.
Obtaining information about a driver and data source			
| SQLDataSources()	Core	Current	Returns the list of available data sources.

Table 12. Function list by category (continued)

Function name divided by task	ODBC 3.0	DB2 ODBC	Purpose
SQLDrivers()	Core	No	Returns the list of installed drivers and their attributes (ODBC 2.0). This function is implemented within the ODBC driver manager and is therefore not applicable within the DB2 UDB for z/OS ODBC environment.
SQLGetFunctions()	Core	Current	Returns supported driver functions.
SQLGetInfo()	Core	Current	Returns information about a specific driver and data source.
SQLGetTypeInfo()	Core	Current	Returns information about supported data types.
Setting and retrieving driver attributes			
SQLGetConnectAttr()	Core	Current	Returns the value of a connection attribute.
SQLGetConnectOption()	Depr	Depr	Returns the value of a connection attribute.
SQLGetEnvAttr()	Core	Current	Returns the value of an environment attribute.
SQLGetStmtAttr()	Core	Current	Returns the value of a statement attribute.
SQLGetStmtOption()	Depr	Depr	Returns the value of a statement attribute.
SQLSetConnectAttr()	Core	Current	Sets a connection attribute.
SQLSetConnectOption()	Depr	Depr	Sets a connection attribute.
SQLSetEnvAttr()	Core	Current	Sets an environment attribute.
SQLSetStmtAttr()	Core	Current	Sets a statement attribute.
SQLSetStmtOption()	Depr	Depr	Sets a statement attribute.
Setting and retrieving descriptor fields			
SQLCopyDesc()	Core	No	Copies descriptor fields.
SQLGetDescField()	Core	No	Returns the value or current setting of a single descriptor field.
SQLGetDescRec()	Core	No	Returns the values or current settings of multiple descriptor fields.
SQLSetDescField()	Core	No	Sets the value or setting for a single descriptor field.
SQLSetDescRec()	Core	No	Sets the values or settings for multiple descriptor fields.
Preparing SQL requests			
SQLAllocStmt()	Depr	Depr	Allocates a statement handle.
SQLBindParameter()	Core	Current	Assigns storage for a parameter in an SQL statement (ODBC 2.0).
SQLGetCursorName()	Core	Current	Returns the cursor name that is associated with a statement handle.
SQLParamOptions()	Depr	Current	Specifies the use of multiple values for parameters. In ODBC 3.0, SQLSetStmtAttr() replaces this function.
SQLPrepare()	Core	Current	Prepares an SQL statement for subsequent execution.
SQLSetCursorName()	Core	Current	Specifies a cursor name.
SQLSetParam()	Depr	Depr	Assigns storage for a parameter in an SQL statement (ODBC 2.0). In ODBC 3.0, SQLBindParameter() replaces this function.

Table 12. Function list by category (continued)

Function name divided by task	ODBC 3.0	DB2 ODBC	Purpose
SQLSetScrollOptions()	Depr	No	Sets attributes that control cursor behavior. In ODBC 3.0, SQLGetInfo() and SQLSetStmtAttr() replace this function.
Submitting requests			
SQLDescribeParam() [1]	Core	Current	Returns the description for a specific input parameter in a statement.
SQLExecDirect()	Core	Current	Executes a statement.
SQLExecute()	Core	Current	Executes a prepared statement.
SQLNativeSql()	Core	Current	Returns the text of an SQL statement as translated by the driver.
SQLNumParams()	Core	Current	Returns the number of parameters in a statement.
SQLParamData()	Core	Current	Used in conjunction with SQLPutData() supplies parameter data at execution time. (Useful for long data values.)
SQLPutData()	Core	Current	Sends part or all of a data value for a parameter. (This function is useful for long data values.)
Retrieving results and information about results			
SQLBindCol()	Core	Current	Assigns storage for a result column and specifies the data type.
SQLBulkOperations()	Lvl 1	No	Performs bulk inserts and bookmark operations.
SQLColAttribute()	Core	Current	Describes attributes of a column in the result set.
SQLColAttributes()	Depr	Depr	Describes attributes of a column in the result set.
SQLDescribeCol()	Core	Current	Describes a column in the result set.
SQLError()	Depr	Depr	Returns additional error or status information.
SQLExtendedFetch()	Depr	Current	Returns multiple result rows.
SQLFetch()	Core	Current	Returns a result row.
SQLFetchScroll()	Core	No	Returns row sets that are specified by absolute or relative position or by bookmark.
SQLGetData()	Core	Current	Returns part or all of one column of one row of a result set. (This function is useful for long data values.)
SQLGetDiagRec()	Core	Current	Returns additional diagnostic information.
SQLGetSQLCA()	No	Current	Returns the SQLCA that is associated with a statement handle.
SQLMoreResults()	Lvl 1	Current	Determines whether more result sets are available and, if so, initializes processing for the next result set.
SQLNumResultCols()	Core	Current	Returns the number of columns in the result set.
SQLRowCount()	Core	Current	Returns the number of rows that are affected by an insert, update, or delete request.
SQLSetColAttributes()	No	Current	Sets attributes of a column in the result set.
SQLSetPos()	Lvl 1	No	Positions a cursor within a fetched block of data.
Handling large objects			

Table 12. Function list by category (continued)

Function name divided by task	ODBC 3.0	DB2 ODBC	Purpose
SQLGetLength()	No	Current	Gets the length, in bytes, of a string that is referenced by a LOB locator.
SQLGetPosition()	No	Current	Gets the position of a string within a source string that is referenced by a LOB locator.
SQLGetSubString()	No	Current	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 from the catalog			
SQLColumnPrivileges()	Lvl 2	Current	Returns a list of columns and associated privileges for a table.
SQLColumns()	Core	Current	Returns the list of column names in specified tables.
SQLForeignKeys()	Lvl 2	Current	Returns a list of column names that comprise foreign keys, if they exist, for a specified table.
SQLPrimaryKeys()	Lvl 1	Current	Returns the list of column names that comprise the primary key for a table.
SQLProcedureColumns()	Lvl 1	Current	Returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures.
SQLProcedures()	Lvl 1	Current	Returns the list of procedure names that are stored in a specific data source.
SQLSpecialColumns()	Core	Current	Returns information about the optimal set of columns that uniquely identifies a row in a specified table, or identifies the columns that are automatically updated when any value in the row is updated by a transaction.
SQLStatistics()	Core	Current	Returns statistics about a single table and the list of indexes that are associated with the table.
SQLTablePrivileges()	Lvl 2	Current	Returns a list of tables and the privileges that are associated with each table.
SQLTables()	Core	Current	Returns the list of table names that are stored in a specific data source.
Terminating a statement			
SQLCancel()	Core	Current	Cancels an SQL statement.
SQLCloseCursor()	Core	Current	Closes a cursor that has been opened on a statement handle.
SQLEndTran()	Core	Current	Commits or rolls back a transaction.
SQLFreeStmt()	Core	Current	Ends statement processing, closes the associated cursor, discards pending results, and, optionally, frees all resources that are associated with the statement handle.
SQLTransact()	Depr	Depr	Commits or rolls back a transaction.
Terminating a connection			
SQLDisconnect()	Core	Current	Closes the connection.
SQLFreeConnect()	Depr	Depr	Releases the connection handle.
SQLFreeEnv()	Depr	Depr	Releases the environment handle.

Table 12. Function list by category (continued)

Function name divided by task	ODBC 3.0	DB2 ODBC	Purpose
SQLFreeHandle()	Core	Current	Releases an environment, connection, statement, or descriptor handle.

The following ODBC 3.0 functions are not supported by DB2 ODBC:

- SQLBrowseConnect().
- SQLBulkOperations().
- SQLCopyDesc(). DB2 ODBC does not support descriptor fields.
- SQLDrivers(). This function is implemented by the ODBC driver manager which does not apply to DB2 ODBC.
- SQLFetchScroll(). DB2 ODBC does not support scrollable cursors. SQLExtendedFetch() retrieves multiple result sets in DB2 ODBC.
- SQLGetDescField(). DB2 ODBC does not support descriptor fields.
- SQLGetDescRec(). DB2 ODBC does not support descriptor fields.
- SQLSetDescField(). DB2 ODBC does not support descriptor fields.
- SQLSetDescRec(). DB2 ODBC does not support descriptor fields.
- SQLSetPos().
- SQLSetScrollOptions(). It is superseded by the SQL_ATTR_CURSOR_TYPE, SQL_ATTR_CONCURRENCY, SQL_KEYSET_SIZE, and SQL_ATTR_ROWSET_SIZE statement attributes.

SQLAllocConnect() - Allocate a connection handle

Purpose

Table 13. SQLAllocConnect() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	Yes

In the current version of DB2 ODBC, SQLAllocHandle() replaces SQLAllocConnect(). See “SQLAllocHandle() - Allocate a handle” on page 72 for more information.

Although DB2 ODBC supports SQLAllocConnect() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLAllocConnect() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLAllocConnect  (SQLHENV          henv,
                               SQLHDBC     FAR  *phdbc);
```

Function arguments

Table 14 lists the data type, use, and description for each argument in this function.

Table 14. SQLAllocConnect() arguments

Data type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle
SQLHDBC *	*phdbc*	output	Pointer to a connection handle

SQLAllocEnv() - Allocate an environment handle

Purpose

Table 15. SQLAllocEnv() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	Yes

In the current version of DB2 ODBC, SQLAllocHandle() replaces SQLAllocEnv(). See “SQLAllocHandle() - Allocate a handle” on page 72 for more information.

Although DB2 ODBC supports SQLAllocEnv() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLAllocEnv() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLAllocEnv      (SQLHENV     FAR   *phenv);
```

Function arguments

Table 16 lists the data type, use, and description for each argument in this function.

Table 16. SQLAllocEnv() arguments

Data type	Argument	Use	Description
SQLHENV *	*phenv*	output	Points to the environment handle that you allocate.

SQLAllocHandle() - Allocate a handle

Purpose

Table 17. SQLAllocHandle() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLAllocHandle() allocates an environment handle, a connection handle, or a statement handle.

Syntax

```
SQLRETURN  SQLAllocHandle    (SQLSMALLINT       HandleType,
                              SQLHANDLE         InputHandle,
                              SQLHANDLE        *OutputHandlePtr);
```

Function arguments

Table 18 lists the data type, use, and description for each argument in this function.

Table 18. SQLAllocHandle() arguments

Data type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	Specifies the type of handle that you want to allocate. Set this argument to one of the following values: • SQL_HANDLE_ENV for an environment handle • SQL_HANDLE_DBC for a connection handle • SQL_HANDLE_STMT for a statement handle
SQLHANDLE	*InputHandle*	input	Specifies the handle from which you allocate the new handle. You set a different value for this argument depending on what type of handle you allocate. Set the *InputHandle* argument to one of the following values: • SQL_NULL_HANDLE (or ignore this argument) if you are allocating an environment handle • To the environment handle if you are allocating a connection handle • To a connection handle if you are allocating a statement handle
SQLHANDLE *	*OutputHandlePtr*	output	Points to the buffer in which SQLAllocHandle() returns the newly allocated handle.

Usage

Use SQLAllocHandle() to allocate an environment handle, connection handles, and statement handles.

- **Allocating an environment handle**

  An environment handle provides access to global information. To request an environment handle in your application, call SQLAllocHandle() with the *HandleType* argument set to SQL_HANDLE_ENV and the *InputHandle* argument set to SQL_NULL_HANDLE. (*InputHandle* is ignored when you allocate an environment handle.) DB2 ODBC allocates the environment handle and passes the value of the associated handle to the **OutputHandlePtr* argument. Your application passes the **OutputHandle* value in all subsequent calls that require an environment handle argument.

When you call SQLAllocHandle() to request an environment handle, the DB2 ODBC 3.0 driver implicitly sets SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC3. See "ODBC 3.0 driver behavior" on page 527 for more information about the SQL_ATTR_ODBC_VERSION environment attribute.

When you allocate an environment handle, the DB2 ODBC 3.0 driver checks the trace keywords in the common section of the DB2 ODBC initialization file. If these keywords are set, DB2 ODBC enables tracing. DB2 ODBC ends tracing when you free the environment handle. See "DB2 ODBC initialization file" on page 49 and Chapter 6, "Problem diagnosis," on page 477 for more information.

The DB2 ODBC 3.0 driver does not support multiple environments. See "Restrictions" on page 75.

- **Allocating a connection handle**

  A connection handle provides access to information such as the valid statement handles on the connection and an indication of whether a transaction is currently open. To request a connection handle, call SQLAllocHandle() with the *HandleType* argument set to SQL_HANDLE_DBC. Set the *InputHandle* argument to the current environment handle. DB2 ODBC allocates the connection handle and returns the value of the associated handle in **OutputHandlePtr*. Pass the **OutputHandlePtr* in all subsequent function calls that require this connection handle as an argument.

  You can allocate multiple connection handles from the context of a single environment handle.

- **Allocating a statement handle**

  A statement handle provides access to statement information, such as messages, the cursor name, and status information about SQL statement processing. To request a statement handle, connect to a data source and then call SQLAllocHandle(). You must allocate a statement handle before you submit SQL statements. In this call, set the *HandleType* argument to SQL_HANDLE_STMT. Set the *InputHandle* argument to the connection handle that is associated with the connection on which you want to execute SQL. DB2 ODBC allocates the statement handle, associates the statement handle with the connection specified, and returns the value of the associated handle in **OutputHandlePtr*. Pass the **OutputHandlePtr* value in all subsequent function calls that require this statement handle as an argument.

  You can allocate multiple statement handles from the context of a single connection handle.

- **Managing handles**

  Your DB2 ODBC applications can allocate multiple connection handles and multiple statement handles at the same time. You can allocate multiple connection handles and make multiple connections only when one or more of the following conditions are true:

  – The connection type is set to coordinated

  – Multiple contexts are enabled

  – You use multiple Language Environment threads

  If you attempt to allocate multiple connection handles when none of these conditions are true, the DB2 ODBC driver will return SQLSTATE **08**001.

  DB2 ODBC 3.0 driver applications can also use the same environment handle, connection handle, or statement handle on multiple threads. DB2 ODBC provides threadsafe access for all handles and function calls. Each connection within a single Language Environment thread maintains its own unit of recovery.

For applications that use more than one Language Environment thread, you must coordinate units of recovery and manage DB2 ODBC resources among Language Environment threads. Your application might behave unpredictably if your application does not perform this task. For example, if you call ODBC functions on different threads for the same connection simultaneously, the order in which these functions are executed at the database is unpredictable. See "Writing multithreaded and multiple-context applications" on page 433 for more information.

Attention: If you call SQLAllocHandle() with **OutputHandlePtr* set to a connection or statement handle that you previously allocated, DB2 ODBC overwrites all information that is associated with that handle. DB2 ODBC does not check whether the handle that is entered in **OutputHandlePtr* is in use, nor does DB2 ODBC check the previous contents of a handle before it overwrites the contents of that handle.

Return codes

After you call SQLAllocHandle(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of each return code value, see "Function return codes" on page 23.

Diagnostics

The way that you retrieve diagnostic information from SQLAllocHandle() depends on what type of handle you allocate. To retrieve diagnostic information from SQLAllocHandle(), you need to consider the following types of errors when you attempt to allocate a handle:

Environment handle allocation errors: When you receive an error while allocating an environment handle, the value to which the *OutputHandlePtr* argument points determines if you can use SQLGetDiagRec() to retrieve diagnostic information. One of the following cases occurs when you fail to allocate an environment handle:
- The *OutputHandlePtr* argument points to SQL_NULL_HENV when SQLAllocHandle() returns SQL_ERROR. In this case, you cannot call SQLGetDiagRec() to retrieve information about this error. Because no handle is associated with the error, you cannot retrieve information about that error.
- The *OutputHandlePtr* argument points to a value other than SQL_NULL_HENV when SQLAllocHandle() returns SQL_ERROR. In this case, the value to which the *OutputHandlePtr* argument points becomes a restricted environment handle. You can use a handle in this restricted state only to call SQLGetDiagRec() to obtain more error information or to call SQLFreeHandle() to free the restricted handle.

Connection or statement handle allocation errors: When you allocate a connection or statement handle, you can retrieve the following types of information:
- When SQLAllocHandle() returns SQL_ERROR, it sets *OutputHandlePtr* to SQL_NULL_HDBC for connection handles or SQL_NULL_HSTMT for statement handles (unless the output argument is a null pointer). Call SQLGetDiagRec() on the environment handle to obtain information about a failed connection handle

allocation. Call SQLGetDiagRec() on a connection handle to obtain information about a failed statement handle allocation.
- When SQLAllocHandle() returns SQL_SUCCESS_WITH_INFO, it returns the allocated handle to *OutputHandlePtr*. To obtain additional information about the allocation, call SQLGetDiagRec() on the handle that you specified in the *InputHandle* argument of SQLAllocHandle().

Table 19 lists each SQLSTATE that this function generates with a description and explanation for each value.

Table 19. SQLAllocHandle() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (SQLAllocHandle() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
08003	Connection is closed.	The *HandleType* argument specifies SQL_HANDLE_STMT, but the connection that is specified in the *InputHandle* argument is not open. The connection process must be completed successfully (and the connection must be open) for DB2 ODBC to allocate a statement handle.
| **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
| **58**004	Unexpected system failure.	This could be a failure to establish the association with the DB2 UDB for z/OS subsystem or any other system-related error.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message that SQLGetDiagRec() returns in the buffer that the *MessageText* argument describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is unable to allocate memory for the specified handle.
HY009	Invalid use of a null pointer.	The *OutputHandlePtr* argument specifies a null pointer
HY013	Unexpected memory handling error.	The *HandleType* argument specifies SQL_HANDLE_DBC or SQL_HANDLE_STMT, 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 that is indicated by the *HandleType* argument has been reached.
HY092	Option type out of range.	The *HandleType* argument does not specify one of the following values: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT

Restrictions

The DB2 ODBC 3.0 driver does not support multiple environments; you can allocate only one active environment at any time. If you call SQLAllocHandle() to allocate more environment handles, this function returns the original environment handle and SQL_SUCCESS. The DB2 ODBC driver keeps an internal count of these environment requests. You must call SQLFreeHandle() on the environment handle for each time that you successfully request an environment handle. The last successful SQLFreeHandle() call that you make on the environment handle frees the DB2 ODBC 3.0 driver environment. This behavior ensures that an ODBC application does not prematurely deallocate the driver environment. The DB2 ODBC 2.0 driver and DB2 ODBC 3.0 driver behave consistently in this situation.

Example

Refer to “DSN8O3VP sample application” on page 531 or DSN8O3VP in the DSN810.SDSNSAMP data set.

Related functions

The following functions relate to SQLAllocHandle() calls. Refer to the descriptions of these functions for more information about how you can use SQLAllocHandle() in your applications.

- “SQLFreeHandle() - Free a handle” on page 190
- “SQLGetDiagRec() - Get multiple field settings of diagnostic record” on page 221
- “SQLSetConnectAttr() - Set connection attributes” on page 346
- “SQLSetEnvAttr() - Set environment attribute” on page 360
- “SQLSetStmtAttr() - Set statement attributes” on page 367

SQLAllocStmt() - Allocate a statement handle

Purpose

Table 20. SQLAllocStmt() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	Yes

In the current version of DB2 ODBC, SQLAllocHandle() replaces SQLAllocStmt(). See “SQLAllocHandle() - Allocate a handle” on page 72 for more information.

Although DB2 ODBC supports SQLAllocStmt() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLAllocStmt() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLAllocStmt     (SQLHDBC          hdbc,
                              SQLHSTMT    FAR  *phstmt);
```

Function arguments

Table 21 lists the data type, use, and description for each argument in this function.

Table 21. SQLAllocStmt() arguments

Data type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Specifies the connection handle
SQLHSTMT *	*phstmt*	output	Points to the newly allocated statement handle

SQLBindCol() - Bind a column to an application variable

Purpose

Table 22. SQLBindCol() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

Use SQLBindCol() to associate, or *bind*, columns in a result set with the following elements of your application:

- Application variables or arrays of application variables (storage buffers) for all C data types. When you bind columns to application variables, data is transferred from the DBMS to the application when you call SQLFetch() or SQLExtendedFetch(). This transfer converts data from an SQL type to any supported C type variable that you specify in the SQLBindCol() call. For more information about data conversion, see “Data types and data conversion” on page 24.
- A LOB locator, for LOB columns. When you bind to LOB locators, the locator, not the data itself, is transferred from the DBMS to the application when you call SQLFetch(). A LOB locator can represent the entire data or a portion of the data.

Call SQLBindCol() once for each column in the result set that your application must retrieve.

Usually you call SQLPrepare(), SQLExecDirect(), or a schema function before you call SQLBindCol(). After you call SQLBindCol(), you usually call SQLFetch() or SQLExtendedFetch(). You might need to obtain column attributes before you call SQLBindCol(). To obtain these attributes, call SQLDescribeCol() or SQLColAttribute().

Syntax

```
SQLRETURN   SQLBindCol       (SQLHSTMT         hstmt,
                              SQLUSMALLINT     icol,
                              SQLSMALLINT      fCType,
                              SQLPOINTER       rgbValue,
                              SQLINTEGER       cbValueMax,
                              SQLINTEGER  FAR  *pcbValue);
```

Function arguments

Table 23 lists the data type, use, and description for each argument in this function.

Table 23. SQLBindCol() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle on which results are returned.
SQLUSMALLINT	*icol*	input	Specifies the number that identifies the column you bind. Columns are numbered sequentially, from left to right, starting at 1.

Table 23. SQLBindCol() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*fCType*	input	The C data type for column number *icol* 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_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_WCHAR The supported data types are based on the data source to which you are connected. Specifying SQL_C_DEFAULT causes data to be transferred to its default C data type. See Table 4 on page 25 for more information.
SQLPOINTER	*rgbValue*	output (deferred)	Points to a buffer (or an array of buffers when you use the SQLExtendedFetch() function) where DB2 ODBC stores the column data or the LOB locator when you fetch data from the bound column. If the *rgbValue* argument is null, the column is unbound.
SQLINTEGER	*cbValueMax*	input	Specifies the size of the *rgbValue* buffer in bytes that are available to store the column data or the LOB locator. If the *fCType* argument denotes a binary or character string (either single-byte or double-byte) or is SQL_C_DEFAULT, *cbValueMax* must be greater than 0, or an error occurs. If the *fCType* argument denotes other data types, the *cbValueMax* argument is ignored.
SQLINTEGER *	*pcbValue*	output (deferred)	Pointer to a value (or array of values) that indicates the number of bytes that DB2 ODBC has available to return in the *rgbValue* buffer. If *fCType* is a LOB locator, the size of the locator, not the size of the LOB data, is returned. SQLFetch() returns SQL_NULL_DATA in this argument if the data value of the column is null. This pointer value can be null. If this pointer is not null, it must be unique for each bound column. SQL_NO_LENGTH can also be returned. See “Usage” on page 80 for more information.

Important: You must ensure the locations that the pointers *rgbValue* and *pcbValue* reference are valid until you call SQLFetch() or SQLExtendedFetch(). For SQLBindCol(), the pointers *rgbValue* and *pcbValue* are deferred outputs, which means that the storage locations to which they point are not updated until you fetch a row from the result set. For example, if you call SQLBindCol() within a local function, you must call SQLFetch() from within the same scope of the function, or you must allocate the *rgbValue* buffer as static or global.

Tip: Place the buffer that the *rgbValue* argument specifies consecutively in memory after the buffer that the *pcbValue* argument specifies for better DB2 ODBC performance for all varying-length data types. See "Usage" for more details.

Usage

Call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators. You generate result sets when you call SQLPrepare(), SQLExecDirect(), SQLGetTypeInfo(), or one of the catalog functions. After you bind columns to a result set, call SQLFetch() to place data from these columns into application storage locations (the locations to which the *rgbValue* and *cbValue* arguments point). If the *fCType* argument specifies a LOB locator, a locator value (not the LOB data itself) is placed in these locations. This locator value references the entire data value in the LOB column at the server.

You can use SQLExtendedFetch() in place of SQLFetch() to retrieve multiple rows from the result set into an array. In this case, the *rgbValue* argument references an array. For more information, see "Retrieving a result set into an array" on page 417 and "SQLExtendedFetch() - Fetch an array of rows" on page 163. You cannot mix calls to SQLExtendedFetch() with calls to SQLFetch() on the same result set.

Obtaining information about the result set: Columns are identified by a number, assigned sequentially from left to right, starting at 1. To determine the number of columns in a result set, call SQLNumResultCols(), or call SQLColAttribute() with the *FieldIdentifier* argument set to SQL_DESC_COUNT.

Call SQLDescribeCol() or SQLColAttribute() to query the attributes (such as data type and length) of a column. (As an alternative, see "Programming hints and tips" on page 470 for information about using SQLSetColAttributes() when you know the format of the result set.) You can then use this information to allocate a storage location with a C data type and length that match the SQL data type an length of the result set column. In the case of LOB data types, you can retrieve a locator instead of the entire LOB value. See "Data types and data conversion" on page 24 for more information about default types and supported conversions.

You can choose which, if any, columns that you want to bind from a result set. For unbound columns, use SQLGetData() instead of, or in conjunction with, SQLBindCol() to retrieve data. Generally, SQLBindCol() is more efficient than SQLGetData(). For a discussion of when to use SQLGetData() instead of SQLBindCol(), refer to "Retrieving data efficiently" on page 473.

During subsequent fetches, you can use SQLBindCol() to change which columns are bound to application variables or to bind previously unbound columns. New binds do not apply to data that you have fetched; these binds are used for the next fetch. To unbind a single column, call SQLBindCol() with the *rgbValue* pointer set to NULL. To unbind all the columns, call SQLFreeStmt() with the *fOption* input set to SQL_UNBIND.

Allocating buffers: Ensure that you allocate enough storage to hold the data that you retrieve. When you allocate a buffer to hold varying-length data, allocate an amount of storage that is equal to the maximum length of data that the column that is bound to this buffer can produce. If you allocate less storage than this maximum, SQLFetch() or SQLExtendedFetch() truncates any data that is larger than the space that you allocated. When you allocate a buffer that holds fixed-length data, DB2 ODBC assumes that the size of the buffer is the length of the C data type. If you specify data conversion, the amount of space that the data requires might change; see “Data types and data conversion” on page 24 for more information.

When you bind a column that is defined as SQL_GRAPHIC, SQL_VARGRAPHIC, or SQL_LONGVARGRAPHIC, you can set the *fCType* argument to SQL_C_DBCHAR, SQL_C_WCHAR, or SQL_C_CHAR. If you set the *fCType* argument to SQL_C_DBCHAR or SQL_C_WCHAR, the data that you fetch into the *rgbValue* buffer is nul-terminated by a double-byte nul-terminator. If you set the *fCType* argument to SQL_C_CHAR, the data that you fetch is not always nul-terminated. In both cases, the length of the *rgbValue* buffer (*cbValueMax*) is in units of bytes, and the value is always a multiple of 2.

When you bind a varying-length column, DB2 ODBC can write to both of the buffers that specified by the *pcbValue* and *rgbValue* arguments in one operation if you allocate these buffers contiguously. The following example illustrates how to allocate these buffers contiguously:

```
struct {  SQLINTEGER  pcbValue;
          SQLCHAR     rgbValue[MAX_BUFFER];
       } column;
```

When the *pcbValue* and *rgbValue* arguments are contiguous, SQL_NO_TOTAL is returned in the *pcbValue* argument if your bind meets **all** of the following conditions:
- The SQL type is a varying-length type.
- The column type is NOT NULLABLE.
- String truncation occurred.

Handling data truncation: If SQLFetch() or SQLExtendedFetch() truncates data, it returns SQL_SUCCESS_WITH_INFO and set the *pcbValue* argument to a value that represents the amount of space (in bytes) that the full data requires.

Truncation is also affected by the SQL_ATTR_MAX_LENGTH statement attribute (which is used to limit the amount of data that your application returns). You can disable truncation warnings with the following procedure:
1. Call SQLSetStmtAttr().
   - Set the *Attribute* argument to SQL_ATTR_MAX_LENGTH.
   - Point the *ValuePtr* argument to a buffer that contains the value for the maximum length, in bytes, of varying-length columns that you want to receive.
2. Allocate the *rgbValue* argument on your SQLBindCol() call as a buffer that is the same size (plus the nul-terminator) as you set for the value of the SQL_ATTR_MAX_LENGTH statement attribute.

If the column data is larger than the maximum length that you specified, the maximum length, not the actual length, is returned in the buffer to which the *pcbValue* argument points. If data is truncated because it exceeds the maximum length that the SQL_ATTR_MAX_LENGTH statement attribute specifies, you receive no warning of this truncation. SQLFetch() and SQLExtendedFetch() return SQL_SUCCESS for data that is truncated in this way.

When you bind a column that holds SQL_ROWID data, you can set the *fCType* argument to SQL_C_CHAR or SQL_C_DEFAULT. The data that you fetch into the buffer that the *rgbValue* argument specifies is nul-terminated. The maximum length of a ROWID column in the DBMS is 40 bytes. Therefore, to retrieve this type of data without truncation, you must allocate an *rgbValue* buffer of at least 40 bytes in your application.

Handling encoding schemes: The DB2 ODBC driver determines one of the following encoding schemes for character and graphic data through the settings of the CURRENTAPPENSCH keyword (which appears in the initialization file) and the *fCType* argument (which you specify in SQLBindCol() calls).

- The ODBC driver places EBCDIC data into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = EBCDIC is specified in the initialization file, or the CURRENTAPPENSCH keyword is not specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLBindCol() call.
- The ODBC driver places Unicode UCS-2 data into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = UNICODE is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_WCHAR in the SQLBindCol() call.
- The ODBC driver places Unicode UTF-8 data into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = UNICODE is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR in the SQLBindCol() call.
- The ODBC driver places ASCII data into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = ASCII is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLBindCol() call.

For more information about encoding schemes, see “Handling application encoding schemes” on page 443.

Retrieved UTF-8 data is terminated by a single-byte nul-terminator, where as retrieved UCS-2 data is terminated by a double-byte nul-terminator.

Binding LOB columns: You generally treat LOB locators like any other data type, but when you use LOB locators the following differences apply:

- The server generates locator values when you fetch from a column that is bound to the LOB locator C data type and passes only the locator, not the data, to the application.
- When you call SQLGetSubString() to define a locator on a portion of another LOB, the sever generates a new locator and transfers only the locator to the application.
- The value of a locator is valid only within the current transaction. You cannot store a locator value and use it beyond the current transaction, even if you specify the WITH HOLD attribute when you define the cursor that you use to fetch the locator.
- You can use the FREE LOCATOR statement to free a locator before the end of a transaction.
- When your application receives a locator, you can use SQLGetSubString() to either receive a portion of the LOB value or to generate another locator that represents a portion of the LOB value. You can also use locator values as input for a parameter marker (with the SQLBindParameter() function).

A LOB locator is not a pointer to a database position; rather, it is a reference to a LOB value, a snapshot of that LOB value. The current position of the cursor and the row from which the LOB value is extracted are not associated. Therefore, even after the cursor moves to a different row, the LOB locator (and thus the value that it represents) can still be referenced.

- With locators, you can use SQLGetPosition() and SQLGetLength() with SQLGetSubString() to define a substring of a LOB value.

You can bind a LOB column to one of the following data types:
- A storage buffer (to hold the entire LOB data value)
- A LOB locator (to hold the locator value only)

The most recent bind column function call determines the type of binding that is in effect.

Return codes

After you call SQLBindCol(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 24 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 24. SQLBindCol() SQLSTATEs

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY002	Invalid column number.	The specified value for the *icol* argument is less than 0 or greater than the number of columns in the result set.
HY003	Program type out of range.	The *fCType* argument is not a valid data type or SQL_C_DEFAULT.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The specified value for the *cbValueMax* argument is less than 0.
HYC00	Driver not capable.	DB2 ODBC does not support the value that the *fCType* argument specifies.
Important: Additional diagnostic messages that relate to the bound columns might be reported at fetch time.		

Restrictions

None.

Example

See Figure 15 on page 168 and “Binding result set columns to retrieve UCS-2 data” on page 449.

Related functions

The following functions relate to SQLBindCol() calls. Refer to the descriptions of these functions for more information about how you can use SQLBindCol() in your applications.

- “SQLExtendedFetch() - Fetch an array of rows” on page 163
- “SQLFetch() - Fetch the next row” on page 171

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator

Purpose

Table 25. SQLBindParameter() specifications

ODBC	X/OPEN CLI	ISO CLI
2.0	No	No

SQLBindParameter() associates, or *binds*, parameter markers in an SQL statement to the following objects:

- All C type application variables or arrays of C type application variables (storage buffers). For application variables, data is transferred from your application to the DBMS when you call SQLExecute() or SQLExecDirect(). This transfer converts data from the C type of the application variable to the SQL type that you specify in the SQLBindParameter() call. For more information about data conversion, see "Data types and data conversion" on page 24.
- SQL LOB type LOB locators. For LOB data types, you transfer a LOB locator value (not the LOB data itself) to the server when you execute an SQL statement.

SQLBindParameter() also binds application storage to a parameter in a stored procedure CALL statement. In this type of bind, parameters can be input, output, or both input and output parameters.

Syntax

```
SQLRETURN  SQL_API SQLBindParameter(
                              SQLHSTMT          hstmt,
                              SQLUSMALLINT      ipar,
                              SQLSMALLINT       fParamType,
                              SQLSMALLINT       fCType,
                              SQLSMALLINT       fSqlType,
                              SQLUINTEGER       cbColDef,
                              SQLSMALLINT       ibScale,
                              SQLPOINTER        rgbValue,
                              SQLINTEGER        cbValueMax,
                              SQLINTEGER   FAR *pcbValue);
```

Function arguments

Table 26 lists the data type, use, and description for each argument in this function.

Table 26. SQLBindParameter() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle of the statement you bind.
SQLUSMALLINT	*ipar*	input	Specifies the parameter marker number, which are ordered sequentially left to right, starting at 1.

Table 26. SQLBindParameter() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*fParamType*	input	Specifies the type of parameter. You can specify the following types of parameters: • 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 *rgbValue* buffer must contain valid input data values; the *pcbValue* buffer must contain the corresponding length value, in bytes, or SQL_NTS, SQL_NULL_DATA, or (if the value should be sent using the SQLParamData() and SQLPutData() functions) 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 *rgbValue* buffer must contain valid input data values; the *pcbValue* buffer must contain the corresponding length value, in bytes, or SQL_NTS, SQL_NULL_DATA, or, if the value should be sent using 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 *rgbValue* and *pcbValue*, unless both are null pointers, in which case the output data is discarded.
SQLSMALLINT	*fCType*	input	Specifies the 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_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_WCHAR Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type to the type indicated in *fSqlType*.

Table 26. SQLBindParameter() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*fSqlType*	input	Specifies the SQL data type of the parameter. The supported types are: • 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_ROWID • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC **Restriction:** SQL_BLOB_LOCATOR, SQL_CLOB_LOCATOR, and SQL_DBCLOB_LOCATOR are application related concepts and do not map to a data type for column definition during a CREATE TABLE.
SQLUINTEGER	*cbColDef*	input	Specifies the precision of the corresponding parameter marker. The meaning of this precision depends on what data type the *fSqlType* argument denotes: • For a binary or single-byte character string (for example, SQL_CHAR, SQL_BINARY), this is the maximum length in bytes for this parameter marker. • For a double-byte character string (for example, SQL_GRAPHIC), this is the maximum length in double-byte characters for this parameter. • For SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision. • For SQL_ROWID, this must be set to 40, the maximum length in bytes for this data type. Otherwise, an error is returned. • Otherwise, this argument is ignored.
SQLSMALLINT	*ibScale*	input	Specifies the scale of the corresponding parameter if the *fSqlType* argument is SQL_DECIMAL or SQL_NUMERIC. If the *fSqlType* argument specifies 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 the values for the *fSqlType* argument that are mentioned here, the *ibScale* argument is ignored.

Table 26. SQLBindParameter() arguments (continued)

Data type	Argument	Use	Description
SQLPOINTER	*rgbValue*	input (deferred), output (deferred), or input (deferred) and output (deferred)	The following characteristics apply to the *rgbValue* argument depending on whether it is an input argument, an output argument, or both: • As an input argument (when the *fParamType* argument specifies SQL_PARAM_INPUT, or SQL_PARAM_INPUT_OUTPUT), *rgbValue* exhibits the following behavior: At execution time, if the *pcbValue* argument does not contain SQL_NULL_DATA or SQL_DATA_AT_EXEC, then *rgbValue* points to a buffer that contains the actual data for the parameter. If the*pcbValue* argument contains SQL_DATA_AT_EXEC, *rgbValue* is an application-defined 32-bit value that is associated with this parameter. This 32-bit value is returned to the application using a subsequent SQLParamData() call. If SQLParamOptions() is called to specify multiple values for the parameter, then *rgbValue* is a pointer to an input buffer array of *cbValueMax* bytes. • As an output argument (when the *fParamType* argument specifies SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT), the *rgbValue* argument points to the buffer where the output parameter value of the stored procedure is stored. If the *fParamType* argument is set to SQL_PARAM_OUTPUT, and both the *rgbValue* argument and the *pcbValue* argument specify null pointers, then the output parameter value or the return value from the stored procedure call is discarded.
SQLINTEGER	*cbValueMax*	input	For character and binary data, the *cbValueMax* argument specifies the size, in bytes, of the buffer that the *rgbValue* argument indicates. If this buffer is a single element, this value specifies the size of that element. If this buffer is an array, the value specifies the size of each element in that array. (Call SQLParamOptions() to specify multiple values for each parameter.) For non-character and non-binary data, this argument is ignored. This length is assumed to be the length that is associated with the C data type in these cases. For output parameters, the *cbValueMax* argument is used to determine whether to truncate character or binary output data. Data is truncated in the following manner: • For character data, if the number of bytes available to return is greater than or equal to the value that the *cbValueMax* argument specifies, the data in the buffer to which the *rgbValue* argument points is truncated. This data is truncated to a length, in bytes, that is equivalent to the value that the *cbValueMax* argument specifies minus one byte. Truncated character data is nul-terminated (unless nul-termination has been turned off). • For binary data, if the number of bytes available to return is greater than the value that the *cbValueMax* argument specifies, the data to which the *rgbValue* argument points is truncated. This data is truncated to a length, in bytes, that is equivalent to the value that the *cbValueMax* argument specifies.

Table 26. SQLBindParameter() arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER *	*pcbValue*	input (deferred), output (deferred), or input (deferred) and output (deferred)	The following characteristics apply to the *pcbValue* argument depending on whether it is an input argument, an output argument, or both: • As an input argument (when the *fParamType* argument specifies SQL_PARAM_INPUT, or SQL_PARAM_INPUT_OUTPUT), the *pcbValue* argument points to the buffer that contains the length, in bytes, of the parameter marker value (when the statement is executed) to which the *rgbValue* argument points. To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA. If the *fCType* argument specifies SQL_C_CHAR or SQL_C_WCHAR, the buffer to which the *pcbValue* argument points must contain either the exact length (in bytes) of the data or SQL_NTS for nul-terminated strings. If the *fCType* argument indicates character data (explicitly, or implicitly with SQL_C_DEFAULT), and the *pcbValue* argument is set to NULL, it is assumed that the application always provides a nul-terminated string in the buffer to which the *rgbValue* argument points. This null setting also implies that the parameter marker never uses null values. If the *fSqlType* argument indicates a graphic data type and the *fCType* argument is set to SQL_C_CHAR, you cannot set the *pcbValue* argument to NULL or point the *pcbValue* argument to a buffer that holds the value SQL_NTS. In general, for graphic data types, the value this buffer holds is the number of bytes that the double-byte data occupies. Always specify a multiple of 2 for the length of double-byte data. If you specify a value that is odd, an error occurs when the statement is executed. When SQLExecute() or SQLExecDirect() is called, and the *pcbValue* argument points to a value of SQL_DATA_AT_EXEC, the data for the parameter is sent with SQLPutData(). This parameter is referred to as a *data-at-execution* parameter. If you use SQLParamOptions() to specify multiple values for each parameter, the *pcbValue* argument points to an array of SQLINTEGER values. Each element in this array specifies the number of bytes (excluding the nul-terminator) that correspond to elements in the array that the *rgbValue* specifies, or the value SQL_NULL_DATA. • As an output argument (when the *fParamType* argument is set to SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT), the *pcbValue* argument points to one of the following values, after the execution of the stored procedure: – number of bytes available to return in *rgbValue*, excluding the nul-termination character. – SQL_NULL_DATA – SQL_NO_TOTAL if the number of bytes available to return cannot be determined.

Usage

Call SQLBindParameter() to bind parameter markers to application variables. Parameter markers are question mark characters (?) that you place in an SQL statement. When you execute a statement that contains parameter markers, each of these markers is replaced with the contents of a host variable.

SQLBindParameter() essentially extends the capability of the SQLSetParam() function by providing the following functionality:
- Can specify whether a parameter is input, output, or both input and output, which is necessary to handle parameters for stored procedures properly.
- Can specify 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. See "SQLParamOptions() - Specify an input array for a parameter" on page 304 and "Using arrays to pass parameter values" on page 414 for more information about using arrays.

Use SQLBindParameter() to bind a parameter marker to one of the following sources:
- An application variable.
- A LOB value from the database server (by specifying a LOB locator).

Binding a parameter marker to an application variable: You must bind a variable to each parameter marker in an SQL statement before you execute that statement. In SQLBindParameter(), the *rgbValue* argument and the *pcbValue* argument are deferred arguments. The storage locations you provide for these arguments must be valid and contain input data values when you execute the bound statement. This requirement means that you must follow **one** of the following guidelines:
- Keep calls to SQLExecDirect() or SQLExecute() in the same procedure scope as calls to SQLBindParameter().
- Dynamically allocate storage locations that you use for input or output parameters.
- Statically declare storage locations that you use for input or output parameters.
- Globally declare storage locations that you use for input or output parameters.

Binding a parameter marker to a LOB locator: When you bind LOB locators to parameter markers the database server supplies the LOB value. Your application transfers only the LOB locator value across the network.

With LOB locators, you can use SQLGetSubString(), SQLGetPosition(), or SQLGetLength(). SQLGetSubString() can return either another locator or the data itself. All locators remain valid until the end of the transaction in which you create them (even when the cursor moves to another row), or until you issue the FREE LOCATOR statement.

Obtaining information about the result set: You can call SQLBindParameter() before SQLPrepare() if you know what columns appear in the result set. Otherwise, if you do not know what columns appear in the result set, you must obtain column attributes after you prepare your query statement.

You reference parameter markers by number, which the *ipar* argument in SQLBindParameter() represents. Parameter markers are numbered sequentially from left to right, starting at 1.

Specifying the parameter type: The *fParamType* 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 can still choose to specify the nature of input or output more exactly on the SQLBindParameter() to follow a more rigorous coding style. When you set the *fParamType* argument, consider the following DB2 ODBC behaviors:

- If an application cannot determine the type of a parameter in a procedure call, set the *fParamType* argument to SQL_PARAM_INPUT; if the data source returns a value for the parameter, DB2 ODBC 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 ODBC sets the buffer that the *pcbValue* argument specifies 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 is processed. If the *rgbValue* and *pcbValue* arguments are both null pointers, DB2 ODBC discards the output value. If the data source does not return a value for an output parameter, DB2 ODBC sets the *pcbValue* buffer to SQL_NULL_DATA.
- When the *fParamType* argument 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. Because the *rgbValue* and *pcbValue* arguments are deferred arguments, you must keep either the SQLExecDirect() or the SQLExecute() call in the same procedure scope as the SQLBindParameter() calls, or the argument values for *rgbValue* and *pcbValue* must be dynamically allocated or statically or globally declared.

  Similarly, if the *fParamType* argument is set to SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT, the buffers that the *rgbValue* and *pcbValue* arguments specify must remain valid until the CALL statement is executed.

Unbinding parameter markers: All parameters that SQLBindParameter() binds remain bound until you perform one of the following actions:

- Call SQLFreeHandle() with the *HandleType* argument set to SQL_HANDLE_STMT.
- Call SQLFreeStmt() with the *fOption* argument set to SQL_RESET_PARAMS.
- Call SQLBindParameter() again for the same parameter *ipar* number.

After an SQL statement is executed, and the results processed, you might want to reuse the statement handle to execute a different SQL statement. If the parameter marker specifications are different (number of parameters, length, or type), you should call SQLFreeStmt() with SQL_RESET_PARAMS to reset or clear the parameter bindings.

The C buffer data type given by *fCType* must be compatible with the SQL data type indicated by *fSqlType*, or an error occurs.

Specifying data-at-execution parameters: An application can pass the value for a parameter either in the *rgbValue* buffer or with one or more calls to SQLPutData(). In calls to SQLPutData(), these parameters are data-at-execution parameters. The application informs DB2 ODBC of a data-at-execution parameter by placing the SQL_DATA_AT_EXEC value in the *pcbValue* buffer. It sets the *rgbValue* input argument to a 32-bit value which is returned on a subsequent SQLParamData() call and can be used to identify the parameter position.

Because the data in the variables referenced by *rgbValue* and *pcbValue* is not verified until the statement is executed, data content or format errors are not detected or reported until SQLExecute() or SQLExecDirect() is called.

Allocating buffers: For character and binary C data, the *cbValueMax* argument specifies the length (in bytes) of the *rgbValue* buffer if it is a single element; or, if the application calls SQLParamOptions() to specify multiple values for each parameter, the *cbValueMax* argument specifies the length (in bytes) of *each* element in the *rgbValue* array, **including** the nul-terminator. If the application specifies multiple values, *cbValueMax* is used to determine the location of values in the *rgbValue* array. For all other types of C data, the *cbValueMax* argument is ignored.

You can pass the value for a parameter with either the buffer that the *rgbValue* argument specifies or one or more calls to SQLPutData(). In calls to SQLPutData(), these parameters are data-at-execution parameters. The application informs DB2 ODBC of a data-at-execution parameter by placing the SQL_DATA_AT_EXEC value in the *pcbValue* buffer. It sets the *rgbValue* input argument to a 32-bit value which is returned on a subsequent SQLParamData() call and can be used to identify the parameter position.

If the *fSqlType* argument is SQL_ROWID, the value for the *cbColDef* argument must be set to 40, which is the maximum length (in bytes) for a ROWID data type. If the *cbColDef* argument is not set to 40, you will receive one of the following SQLSTATEs:
- SQLSTATE **22**001 when the *cbColDef* argument is less than 40
- SQLSTATE **HY**104 when the *cbColDef* argument is greater than 40

Handling encoding schemes: When SQLBindParameter() is used to bind an application variable to an output parameter for a stored procedure, DB2 ODBC can provide some performance enhancement if the *rgbValue* buffer is placed consecutively in memory after the *pcbValue* buffer. For example:

```
struct {  SQLINTEGER  pcbValue;
          SQLCHAR     rgbValue[MAX_BUFFER];
       } column;
```

| The DB2 ODBC driver determines one of the following encoding schemes for
| character and graphic data through the settings of the CURRENTAPPENSCH
| keyword (which appears in the initialization file) and the *fCType* argument (which
| you specify in SQLBindParameter() calls):

| • The ODBC driver places EBCDIC data into application variables when both of
| the following conditions are true:
| – CURRENTAPPENSCH = EBCDIC is specified in the initialization file, or the
| CURRENTAPPENSCH keyword is not specified in the initialization file.
| – The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the
| SQLBindParameter() call.
| • The ODBC driver places Unicode UCS-2 data into application variables when
| both of the following conditions are true:
| – CURRENTAPPENSCH = UNICODE is specified in the initialization file.
| – The *fCType* argument specifies SQL_C_WCHAR in the SQLBindParameter()
| call.
| • The ODBC driver places Unicode UTF-8 data into application variables when
| both of the following conditions are true:
| – CURRENTAPPENSCH = UNICODE is specified in the initialization file.

  - The *fCType* argument specifies SQL_C_CHAR in the SQLBindParameter() call.
- The ODBC driver places ASCII data into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = ASCII is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLBindParameter() call.

For more information about encoding schemes, see "Handling application encoding schemes" on page 443.

Return codes

After you call SQLBindParameter(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 27 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 27. SQLBindParameter() SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The conversion from the data value identified by the *fCType* argument to the data type that is identified by the *fSqlType* argument, is not a meaningful conversion. (For example, a conversion from SQL_C_TYPE_DATE to SQL_DOUBLE is not meaningful.)
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	An unrecoverable system error occurs.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY003	Program type out of range.	The *fCType* argument is not a valid data type or SQL_C_DEFAULT.
HY004	Invalid SQL data type.	The specified value for the *fSqlType* argument is not a valid SQL data type.
HY009	Invalid use of a null pointer.	The argument *OutputHandlePtr* is a null pointer.
HY010	Function sequence error.	The function is called after SQLExecute() or SQLExecDirect() return SQL_NEED_DATA, but data is not sent for all data-at-execution parameters.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The specified value for the *cbValueMax* argument is less than 0.
HY093	Invalid parameter number.	The specified value for the *ipar* argument is less than 1.

Table 27. SQLBindParameter() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY104	Invalid precision or scale value.	This SQLSTATE is returned for one or more of the following reasons: • The specified value for the *fSqlType* argument is either SQL_DECIMAL or SQL_NUMERIC, and the specified value for the *cbColDef* argument is less than 1. • The specified value for the *fCType* argument is SQL_C_TYPE_TIMESTAMP, the value for the *fSqlType* argument is either SQL_CHAR or SQL_VARCHAR, and the value for the *ibScale* argument is less than 0 or greater than 6.
HY105	Invalid parameter type.	The *fParamType* argument does not specify one of the following values: • SQL_PARAM_INPUT • SQL_PARAM_OUTPUT • SQL_PARAM_INPUT_OUTPUT
HYC00	Driver not capable.	This SQLSTATE is returned for one or more of the following reasons: • DB2 ODBC or the data source does not support the conversion that is specified by the combination of the specified value for the *fCType* argument and the specified value for the *fSqlType* argument. • The specified value for the *fSqlType* argument is not supported by either DB2 ODBC or the data source.

Restrictions

A new value for the *pcbValue* argument, SQL_DEFAULT_PARAM, was introduced in ODBC 2.0 to indicate that the procedure should use the default value of a parameter, rather than a value sent from the application. Because DB2 stored procedure arguments do not use default values, specification of SQL_DEFAULT_PARAM for the *pcbValue* argument results in an error when the CALL statement is executed. This error occurs because the SQL_DEFAULT_PARAM value is considered an invalid length.

ODBC 2.0 also introduced the SQL_LEN_DATA_AT_EXEC(*length*) macro to be used with the *pcbValue* argument. The macro specifies the sum total length of all character C data or all binary C data that is sent with the subsequent SQLPutData() calls. Because the DB2 ODBC driver does not need this information, the macro is not needed. To check if the driver needs this information, call SQLGetInfo() with the *InfoType* argument set to SQL_NEED_LONG_DATA_LEN. The DB2 ODBC driver returns 'N' to indicate that this information is not needed by SQLPutData().

Example

Figure 8 on page 95 shows an application that binds a variety of data types to a set of parameters. For additional examples see Appendix F, "Example DB2 ODBC code," on page 531 and Figure 57 on page 450.

```
/* ... */
    SQLCHAR        stmt[] =
    "INSERT INTO PRODUCT VALUES (?, ?, ?, ?, ?)";

    SQLINTEGER     Prod_Num[NUM_PRODS] = {
        100110, 100120, 100210, 100220, 100510, 100520, 200110,
        200120, 200210, 200220, 200510, 200610, 990110, 990120,
        500110, 500210, 300100
    };

    SQLCHAR        Description[NUM_PRODS][257] = {
        "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[NUM_PRODS] = {
        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[NUM_PRODS][3] = {
        "  ", "  ", "  ", "  ", "  ", "  ", "  ", "  ", "  ",
        "  ", "  ", "  ", "  ", "  ", "kg", "kg", "m"
    };
    SQLCHAR        Combo[NUM_PRODS][2] = {
        "N", "N", "N", "N", "N", "N", "N", "N", "N",
        "N", "N", "N", "Y", "Y", "N", "N", "N"
    };
    SQLUINTEGER    pirow = 0;
/* ... */
```

Figure 8. An application that binds data types to parameters (Part 1 of 2)

```
    /* Prepare the statement */
    rc = SQLPrepare(hstmt, stmt, SQL_NTS);

    rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER,
                          0, 0, Prod_Num, 0, NULL);

    rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
                     257, 0, Description, 257, NULL);

    rc = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_DOUBLE, SQL_DECIMAL,
                          10, 2, UPrice, 0, NULL);

    rc = SQLBindParameter(hstmt, 4, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          3, 0, Units, 3, NULL);

    rc = SQLBindParameter(hstmt, 5, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          2, 0, Combo, 2, NULL);

    rc = SQLParamOptions(hstmt, NUM_PRODS, &pirow);

    rc = SQLExecute(hstmt);
    printf("Inserted %ld Rows\n", pirow);
/* ... */
```

Figure 8. An application that binds data types to parameters (Part 2 of 2)

Related functions

The following functions relate to SQLBindParameter() calls. Refer to the descriptions of these functions for more information about how you can use SQLBindParameter() in your applications.

- “SQLExecDirect() - Execute a statement directly” on page 154
- “SQLExecute() - Execute a statement” on page 160
- “SQLParamData() - Get next parameter for which a data value is needed” on page 301
- “SQLParamOptions() - Specify an input array for a parameter” on page 304
- “SQLPutData() - Pass a data value for a parameter” on page 335

SQLCancel() - Cancel statement

Purpose

Table 28. SQLCancel() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

Use SQLCancel() to prematurely terminate a data-at-execution sequence, which is described in “Sending or retrieving long data values in pieces” on page 412.

Syntax

```
SQLRETURN   SQLCancel       (SQLHSTMT         hstmt);
```

Function arguments

Table 29 lists the data type, use, and description for each argument in this function.

Table 29. SQLCancel() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle

Usage

After SQLExecDirect() or SQLExecute() returns SQL_NEED_DATA to solicit values for data-at-execution parameters, you can use SQLCancel() to cancel the data-at-execution sequence. You can call SQLCancel() any time before the final SQLParamData() in the sequence. After you cancel this sequence, you can call SQLExecute() or SQLExecDirect() to re-initiate the data-at-execution sequence.

If you call SQLCancel() on an statement handle that is not associated with a data-at-execution sequence, SQLCancel() has the same effect as SQLFreeHandle() with the *HandleType* set to SQL_HANDLE_STMT. You should not call SQLCancel() to close a cursor; rather, use SQLCloseCursor() to close cursors.

Return codes

After you call SQLCancel(), it returns one of the following values:
- SQL_SUCCESS
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 30 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 30. SQLCancel() SQLSTATEs

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.

Table 30. SQLCancel() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.

Restrictions

DB2 ODBC does not support asynchronous statement execution.

Related functions

The following functions relate to SQLCancel() calls. Refer to the descriptions of these functions for more information about how you can use SQLCancel() in your applications.

- "SQLParamData() - Get next parameter for which a data value is needed" on page 301
- "SQLPutData() - Pass a data value for a parameter" on page 335

SQLCloseCursor() - Close a cursor and discard pending results

Purpose

Table 31. SQLCloseCursor() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLCloseCursor() closes a cursor that has been opened on a statement and discards pending results.

Syntax

```
SQLRETURN  SQLCloseCursor (SQLHSTMT         StatementHandle);
```

Function arguments

Table 32 lists the data type, use, and description for each argument in this function.

Table 32. SQLCloseCursor() arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.

Usage

SQLCloseCursor() closes a cursor that has been opened on a statement and discards pending results. After an application calls SQLCloseCursor(), the application can reopen the cursor by executing a SELECT statement again with the same or different parameter values. When the cursor is reopened, the application uses the same statement handle.

SQLCloseCursor() returns SQLSTATE **24**000 (invalid cursor state) if no cursor is open. Calling SQLCloseCursor() is equivalent to calling the ODBC 2.0 function SQLFreeStmt() with *fOption* argument set to SQL_CLOSE. An exception is that SQLFreeStmt() with SQL_CLOSE has no effect on the application if no cursor is open on the statement, whereas SQLCloseCursor() returns SQLSTATE **24**000 (invalid cursor state).

Return codes

After you call SQLCloseCursor(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 33 on page 100 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 33. SQLCloseCursor() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (SQLCloseCursor() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
24000	Invalid cursor state.	No cursor is open on the statement handle.
HY000	General error.	An error occurred for which no specific SQLSTATE applies. The error message that SQLGetDiagRec() returns in the buffer that the *MessageText* argument specifies, describes this error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is unable to allocate memory that is required execute or complete the function.
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() are called on the statement handle and return SQL_NEED_DATA. SQLCloseCursor() is called before data was sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY013	Unexpected memory handling error.	DB2 ODBC is unable to access memory that is required to support execution or completion of the function.

Restrictions

None.

Example

The following lines of code close the cursor on statement handle `hstmt`:

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

Related functions

The following functions relate to SQLCloseCursor() calls. Refer to the descriptions of these functions for more information about how you can use SQLCloseCursor() in your applications.

- "SQLGetConnectAttr() - Get current attribute setting" on page 196
- "SQLSetConnectAttr() - Set connection attributes" on page 346
- "SQLSetStmtAttr() - Set statement attributes" on page 367

SQLColAttribute() - Get column attributes

Purpose

Table 34. SQLColAttribute() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLColAttribute() returns descriptor information about 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,
                             SQLSMALLINT     ColumnNumber,
                             SQLSMALLINT     FieldIdentifier,
                             SQLPOINTER      CharacterAttributePtr,
                             SQLSMALLINT     BufferLength,
                             SQLSMALLINT     *StringLengthPtr,
                             SQLPOINTER      NumericAttributePtr);
```

Function arguments

Table 35 lists the data type, use, and description for each argument in this function.

Table 35. SQLColAttribute() arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLUSMALLINT	*ColumnNumber*	input	Number of the column you want to be described. Columns are numbered sequentially from left to right, starting at 1. Column zero might not be defined. The DB2 ODBC 3.0 driver does not support bookmarks. See “Restrictions” on page 108.
SQLSMALLINT	*FieldIdentifier*	input	The field in row *ColumnNumber* that is to be returned. See Table 36 on page 102.
SQLPOINTER	*CharacterAttributePtr*	output	Pointer to a buffer in which to return the value in the *FieldIdentifier* field of the *ColumnNumber* row if the field is a character string. Otherwise, this field is ignored.
SQLSMALLINT	*BufferLength*	input	The length, in bytes, of the buffer you specified for the **CharacterAttributePtr* argument, if the field is a character string. Otherwise, this field is ignored.
SQLSMALLINT *	*StringLengthPtr*	output	Pointer to a buffer in which to return the total number of bytes (excluding the nul-termination character) that are available to return in **CharacterAttributePtr*. For character data, if the number of bytes that are available to return is greater than or equal to *BufferLength*, the descriptor information in **CharacterAttributePtr* is truncated to *BufferLength* minus the length (in bytes) of a nul-termination character. DB2 ODBC then nul-terminates the value. For all other types of data, the value of *BufferLength* is ignored, and DB2 ODBC assumes that the size of **CharacterAttributePtr* is 32 bits.

Table 35. SQLColAttribute() arguments (continued)

Data type	Argument	Use	Description
SQLPOINTER	*NumericAttributePtr*	output	Pointer to an integer buffer in which to return the value in the *FieldIdentifier* field of the *ColumnNumber* row, if the field is a numeric descriptor type, such as SQL_DESC_LENGTH. Otherwise, this field is ignored.

Usage

SQLColAttribute() returns information in either **NumericAttributePtr* or **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 ODBC ignores *CharacterAttributePtr*, *BufferLength*, and *StringLengthPtr*. When information is returned in **CharacterAttributePtr*, DB2 ODBC ignores *NumericAttributePtr*.

SQLColAttribute() allows access to the more extensive set of descriptor information that is available in ANSI SQL92 and DBMS vendor extensions. SQLColAttribute() is a more extensible alternative to the SQLDescribeCol() function, but that function can return only one attribute per call.

DB2 ODBC must return a value for each of the descriptor types. If a descriptor type does not apply to a data source, DB2 ODBC returns 0 in **StringLengthPtr* or an empty string in **CharacterAttributePtr* unless otherwise stated.

Table 36 lists the descriptor types that are returned by ODBC 3.0 SQLColAttribute(), along with the ODBC 2.0 SQLColAttributes() attribute values (in parentheses) that were replaced or renamed.

Table 36. SQLColAttribute() field identifiers

Field identifier	Information returned in arguments	Description
SQL_DESC_AUTO_UNIQUE_VALUE (SQL_COLUMN_AUTO_INCREMENT)[1]	*NumericAttributePtr*	Indicates whether the column data type automatically increments. SQL_FALSE is returned in *NumericAttributePtr* for all DB2 SQL data types.
SQL_DESC_BASE_COLUMN_NAME	*CharacterAttributePtr*	The base column name for the set column. If a base column name does not exist (for example, columns that are expressions), this variable contains an empty string.
SQL_DESC_BASE_TABLE_NAME	*CharacterAttributePtr*	The name of the base table that contains the column. If the base table name cannot be defined or is not applicable, this variable contains an empty string.
SQL_DESC_CASE_SENSITIVE (SQL_COLUMN_CASE_SENSITIVE[1]	*NumericAttributePtr*	Indicates if the column data type is case sensitive. Either SQL_TRUE or SQL_FALSE is returned in *NumericAttributePtr*, depending on the data type. Case sensitivity does not apply to graphic data types. SQL_FALSE is returned for non-character data types.
SQL_DESC_CATALOG_NAME (SQL_COLUMN_CATALOG_NAME)[1] (SQL_COLUMN_QUALIFIER_NAME)[1]	*CharacterAttributePtr*	The name of the catalog table that contains the column. An empty string is returned because DB2 ODBC supports two-part naming for a table.

Table 36. SQLColAttribute() field identifiers (continued)

Field identifier	Information returned in arguments	Description
SQL_DESC_CONCISE_TYPE	*CharacterAttributePtr*	The concise data type. For datetime data types, this field returns the concise data type, such as SQL_TYPE_TIME.
SQL_DESC_COUNT (SQL_COLUMN_COUNT)[1]	*NumericAttributePtr*	The number of columns in the result set.
SQL_DESC_DISPLAY_SIZE (SQL_COLUMN_DISPLAY_SIZE)[1]	*NumericAttributePtr*	The maximum number of bytes that are needed to display the data in character form. See Appendix D, "Data conversion," on page 509 for details about the display size of each of the column data types.
SQL_DESC_DISTINCT_TYPE (SQL_COLUMN_DISTINCT_TYPE)[1]	*CharacterAttributePtr*	The distinct type name that is used for a column. If the column is a built-in SQL type and not a distinct type, an empty string is returned. **IBM specific:** This is an IBM-defined extension to the list of descriptor attributes as defined by ODBC.
SQL_DESC_FIXED_PREC_SCALE (SQL_COLUMN_MONEY)[1]	*NumericAttributePtr*	SQL_TRUE if the column has a fixed precision and nonzero scale that are data-source-specific. This value is SQL_FALSE if the column does not have a fixed precision and nonzero scale that are data-source-specific. SQL_FALSE is returned in *NumericAttributePtr* for all DB2 SQL data types.
SQL_DESC_LABEL (SQL_COLUMN_LABEL)[1]	*CharacterAttributePtr*	The column label. If the column does not have a label, the column name or the column expression is returned. If the column is not labeled or named, an empty string is returned.
SQL_DESC_LENGTH	*NumericAttributePtr*	A numeric value that is either the maximum or actual length, in bytes, of a character string or binary data type. This value is the maximum length for a fixed-length data type, or the actual length for a varying-length data type. This value always excludes the nul-termination character that ends the character string.
SQL_DESC_LITERAL_PREFIX	*CharacterAttributePtr*	A VARCHAR(128) record field that contains the character or characters that DB2 ODBC recognizes as a prefix for a literal of this data type. This field contains an empty string if a literal prefix is not applicable to this data type.
SQL_DESC_LITERAL_SUFFIX	*CharacterAttributePtr*	A VARCHAR(128) record field that contains the character or characters that DB2 ODBC recognizes as a suffix for a literal of this data type. This field contains an empty string if a literal prefix is not applicable to this data type.

Table 36. SQLColAttribute() field identifiers (continued)

Field identifier	Information returned in arguments	Description
SQL_DESC_LOCAL_TYPE_NAME	*CharacterAttributePtr*	A VARCHAR(128) record field that contains any localized (native language) name for the data type that might be different from the regular name of the data type. If a localized name does not exist, an empty string is returned. This field is for display purposes only. The character set of the string is location dependent; it is typically the default character set of the server.
SQL_DESC_NAME (SQL_COLUMN_NAME)[1]	*CharacterAttributePtr*	The name of the column specified with *ColumnNumber*. If the column is an expression, the column number is returned. In either case, SQL_DESC_UNNAMED is set to SQL_NAMED. If the column is unnamed or has no alias, an empty string is returned and SQL_DESC_UNNAMED is set to SQL_UNNAMED.
SQL_DESC_NULLABLE (SQL_COLUMN_NULLABLE)[1]	*NumericAttributePtr*	If the column that is identified by *ColumnNumber* can contain null values, SQL_NULLABLE is returned. If the column cannot accept null values, SQL_NO_NULLS is returned.
SQL_DESC_NUM_PREX_RADIX	*NumericAttributePtr*	The precision of each digit in a numeric value. The following values are commonly returned: • If the data type 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 data type 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 nonnumeric data types.

Table 36. SQLColAttribute() field identifiers (continued)

Field identifier	Information returned in arguments	Description
SQL_DESC_OCTET_LENGTH (SQL_COLUMN_LENGTH)[1]	*NumericAttributePtr*	The number of bytes of data that is associated with the column. This is the length in bytes of data that is transferred on the fetch or SQLGetData() for this column if SQL_C_DEFAULT is specified as the C data type. See Appendix D, "Data conversion," on page 509 for details about the length of each of the SQL data types. If the column that is 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 that is identified in *ColumnNumber* is a varying-length character or binary string, (for example, SQL_VARCHAR or SQL_BLOB), the maximum length is returned.
SQL_DESC_PRECISION (SQL_COLUMN_PRECISION)[1]	*NumericAttributePtr*	The precision in units of digits if the column is: • SQL_DECIMAL • SQL_NUMERIC • SQL_DOUBLE • SQL_FLOAT • SQL_INTEGER • SQL_REAL • SQL_SMALLINT If the column is a character SQL data type, the precision that is returned indicates the maximum number of characters that the column can hold. If the column is a graphic SQL data type, the precision indicates the maximum number of double-byte characters that the column can hold. See Appendix D, "Data conversion," on page 509 for information about the precision of each of the SQL data types.
SQL_DESC_SCALE (SQL_COLUMN_SCALE)[1]	*NumericAttributePtr*	The scale attribute of the column. See Appendix D, "Data conversion," on page 509 for information about the precision of each of the SQL data types.
SQL_DESC_SCHEMA_NAME (SQL_COLUMN_OWNER_NAME)[1]	*CharacterAttributePtr*	The schema of the table that contains the column. An empty string is returned; DB2 is not able to determine this attribute.

Table 36. SQLColAttribute() field identifiers (continued)

Field identifier	Information returned in arguments	Description
SQL_DESC_SEARCHABLE (SQL_COLUMN_SEARCHABLE)[1]	*NumericAttributePtr*	Indicates if the column data type is searchable: • SQL_PRED_NONE (SQL_UNSEARCHABLE in ODBC 2.0) if the column cannot be used in a WHERE clause. • SQL_PRED_CHAR (SQL_LIKE_ONLY in ODBC 2.0) if the column can be used in a WHERE clause only with the LIKE predicate. • SQL_PRED_BASIC (SQL_ALL_EXCEPT_LIKE in ODBC 2.0) 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 (SQL_COLUMN_TABLE_NAME)[1]	*CharacterAttributePtr*	The name of the table that contains the column. An empty string is returned; DB2 ODBC cannot determine this attribute.
SQL_DESC_TYPE (SQL_COLUMN_TYPE)[1]	*NumericAttributePtr*	The SQL data type of the column. See Appendix D, "Data conversion," on page 509 for a list of possible data type values that can be returned. For the datetime data types, this field returns the verbose data type, such as SQL_DATETIME.
SQL_DESC_TYPE_NAME (SQL_COLUMN_TYPE_NAME)[1]	*CharacterAttributePtr*	The type of the column (specified in an SQL statement). See Appendix D, "Data conversion," on page 509 for information about each data type.
SQL_DESC_UNNAMED	*NumericAttributePtr*	Returns SQL_NAMED or SQL_UNNAMED. If the SQL_DESC_NAME contains a column name, SQL_NAMED is returned. If the column is unnamed, SQL_UNNAMED is returned.
SQL_DESC_UNSIGNED (SQL_COLUMN_UNSIGNED)[1]	*NumericAttributePtr*	Indicates if the column data type is an unsigned type. SQL_TRUE is returned in *NumericAttributePtr* for all nonnumeric data types. SQL_FALSE is returned for all numeric data types.

Table 36. SQLColAttribute() field identifiers (continued)

Field identifier	Information returned in arguments	Description
SQL_DESC_UPDATABLE (SQL_COLUMN_UPDATABLE)[1]	*NumericAttributePtr*	Indicates if the column data type is a data type that can be updated: • 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. ODBC also defines the following values, however DB2 ODBC does not return these values: – SQL_DESC_UPDATABLE – SQL_UPDT_WRITE

Note:

1. These descriptor values (values for argument *fDescType*) are for the deprecated ODBC 2.0 SQLColAttributes() API. Both SQLColAttribute() and SQLColAttributes() support these values.

Return codes

After you call SQLColAttribute(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 37 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 37. SQLColAttribute() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (SQLColAttribute() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
01004	Data truncated.	The buffer to which the *CharacterAttributePtr* argument points is not large enough to return the entire string value, so the string value was truncated. The length, in bytes, of the untruncated string value is returned in the buffer to which the *StringLengthPtr* argument points. (SQLColumnAttribute() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
07005	The statement did not return a result set.	The statement that is associated with the *StatementHandle* argument did not return a result set. There are no columns to describe.
HY000	General error.	An error occurred for which there is no specific SQLSTATE. The error message that is returned by SQLGetDiagRec() in the buffer to which the *MessageText* argument points, describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory that is required to support execution or completion of the function.

Table 37. SQLColAttribute() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
| **HY**002	Invalid column number.	The value that is specified for the *ColumnNumber* argument is less than 0, or greater than the number of columns in the result set.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • The function is called prior to SQLPrepare() or SQLExecDirect() for the statement handle that the *StatementHandle* argument specifies. • SQLExecute() or SQLExecDirect() is called for the statement handle that the *StatementHandle* argument specifies and returns SQL_NEED_DATA. SQLColAttribute() is called before data is sent for all data-at-execution parameters or columns.
HY090	Invalid string or buffer length.	The value that is specified for the *BufferLength* argument is less than 0.
HY091	Descriptor type out of range.	The value that is specified for the *FieldIdentifier* argument is neither one of the defined values nor an implementation-defined value.
HYC00	Driver not capable.	DB2 ODBC does not support the specified value for the *FieldIdentifier* argument.

Restrictions

ColumnNumber zero might not be defined. The DB2 ODBC 3.0 driver does not support bookmarks.

Example

See Figure 13 on page 134. In this example, SQLColAttribute() retrieves the display length for a column.

Related functions

The following functions relate to SQLColAttribute() calls. Refer to the descriptions of these functions for more information about how you can use SQLColAttribute() in your applications.
- “SQLBindCol() - Bind a column to an application variable” on page 78
- “SQLDescribeCol() - Describe column attributes” on page 131
- “SQLExtendedFetch() - Fetch an array of rows” on page 163
- “SQLFetch() - Fetch the next row” on page 171

SQLColAttributes() - Get column attributes

Purpose

Table 38. SQLColAttributes() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	No	No

In the current version of DB2 ODBC, SQLColAttribute() replaces SQLColAttributes(). See “SQLColAttribute() - Get column attributes” on page 101 for more information.

Although DB2 ODBC supports SQLColAttributes() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLColAttributes() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLColAttributes (SQLHSTMT          hstmt,
                              SQLUSMALLINT      icol,
                              SQLUSMALLINT      fDescType,
                              SQLPOINTER        rgbDesc,
                              SQLSMALLINT       cbDescMax,
                              SQLSMALLINT FAR   *pcbDesc,
                              SQLINTEGER  FAR   *pfDesc);
```

Function arguments

Table 39 lists the data type, use, and description for each argument in this function.

Table 39. SQLColAttributes() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLUSMALLINT	*icol*	input	Column number in the result set (must be between 1 and the number of columns in the result set, inclusive). This argument is ignored when SQL_COLUMN_COUNT is specified.
SQLUSMALLINT	*fDescType*	input	The supported values are described in Table 36 on page 102.
SQLCHAR *	*rgbDesc*	output	Pointer to buffer for string column attributes.
SQLSMALLINT	*cbDescMax*	input	Specifies the length, in bytes, of *rgbDesc* descriptor buffer.
SQLSMALLINT *	*pcbDesc*	output	Actual number of bytes that are returned in *rgbDesc* buffer. If this argument contains a value equal to or greater than the length that is specified in *cbDescMax*, truncation occurred. The column attribute value is then truncated to *cbDescMax* bytes, minus the size of the nul-terminator (or to *cbDescMax* bytes if nul-termination is off).
SQLINTEGER *	*pfDesc*	output	Pointer to an integer that holds the value of numeric column attributes.

SQLColumnPrivileges() - Get column privileges

Purpose

Table 40. SQLColumnPrivileges() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

SQLColumnPrivileges() returns a list of columns and associated privileges for the specified table. The information is returned in an SQL result set, which you can retrieve by using the same functions that you use to process a result set that a query generates.

Syntax

```
SQLRETURN SQLColumnPrivileges  (SQLHSTMT        hstmt,
                                SQLCHAR    FAR  *szCatalogName,
                                SQLSMALLINT     cbCatalogName,
                                SQLCHAR    FAR  *szSchemaName,
                                SQLSMALLINT     cbSchemaName,
                                SQLCHAR    FAR  *szTableName,
                                SQLSMALLINT     cbTableName,
                                SQLCHAR    FAR  *szColumnName,
                                SQLSMALLINT     cbColumnName);
```

Function arguments

Table 41 lists the data type, use, and description for each argument in this function.

Table 41. SQLColumnPrivileges() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLCHAR *	*szCatalogName*	input	Catalog qualifier of a three-part table name. This must be a null pointer or a zero-length string.
SQLSMALLINT	*cbCatalogName*	input	Specifies the length, in bytes, of *szCatalogName*. This must be set to 0.
SQLCHAR *	*szSchemaName*	input	Schema qualifier of table name.
SQLSMALLINT	*cbSchemaName*	input	The length, in bytes, of *szSchemaName*.
SQLCHAR *	*szTableName*	input	Table name.
SQLSMALLINT	*cbTableName*	input	The length, in bytes, of *szTableName*.
SQLCHAR *	*szColumnName*	input	Buffer that can contain a pattern-value to qualify the result set by column name.
SQLSMALLINT	*cbColumnName*	input	The length, in bytes, of *szColumnName*.

Usage

The results are returned as a standard result set that contains the columns listed in Table 42 on page 111. 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. Typically, you call this function after a call to SQLColumns() to determine column privilege information. The application should use the character strings that are returned in the TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns of the SQLColumns() result set as input arguments to this function.

Because calls to SQLColumnPrivileges() frequently result in a complex and thus expensive query to the catalog, used these calls sparingly, and save the results rather than repeat the calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 bytes (which is consistent with SQL92 limits). Because DB2 names are shorter than 128 characters, the application can choose to always set aside 128 characters (plus the nul-terminator) for the output buffer. You can alternatively call SQLGetInfo() with the *InfoType* argument set to each of the following values:

- SQL_MAX_CATALOG_NAME_LEN, to determine the length of TABLE_CAT columns that the connected DBMS supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected DBMS supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected DBMS supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected DBMS supports

Note that the *szColumnName* argument accepts a search pattern. For more information about valid search patterns, see “Input arguments on catalog functions” on page 408.

Although new columns might be added and the names of the existing columns might change in future releases, the position of the current columns will remain unchanged. Table 42 lists the columns in the result set that SQLColumnPrivileges() currently returns.

Table 42. Columns returned by SQLColumnPrivileges()

Column number	Column name	Data type	Description
1	TABLE_CAT	VARCHAR(128)	Always null.
2	TABLE_SCHEM	VARCHAR(128)	Indicates the name of the schema that contains TABLE_NAME.
3	TABLE_NAME	VARCHAR(128) not NULL	Indicates the name of the table or view.
4	COLUMN_NAME	VARCHAR(128) not NULL	Indicates the name of the column of the specified table or view.
5	GRANTOR	VARCHAR(128)	Indicates the authorization ID of the user who granted the privilege.
6	GRANTEE	VARCHAR(128)	Indicates the authorization ID of the user to whom the privilege is granted.

Table 42. Columns returned by SQLColumnPrivileges() (continued)

Column number	Column name	Data type	Description
7	PRIVILEGE	VARCHAR(128)	Indicates the column privilege. This can be: • ALTER • CONTROL • DELETE • INDEX • INSERT • REFERENCES • SELECT • UPDATE Supported privileges are based on the data source to which you are connected. Most IBM RDBMSs do not offer column-level privileges at the column level. DB2 UDB for z/OS and DB2 for VSE & VM support the UPDATE column privilege; each updatable column is receives one row in this result set. For all other privileges for DB2 UDB for z/OS 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.
8	IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users. Either ″YES″ or ″NO″.

The column names that DB2 ODBC uses follow the X/Open CLI CAE specification style. The column types, contents, and order are identical to those that are defined for the SQLColumnPrivileges() result set in ODBC.

If more than one privilege is associated with a column, each privilege is returned as a separate row in the result set.

Return codes

After you call SQLColumnPrivileges(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 43 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 43. SQLColumnPrivileges() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is open on the statement handle.

Table 43. SQLColumnPrivileges() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	The *szTableName* argument is null.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0 and not equal to SQL_NTS.
HYC00	Driver not capable.	DB2 ODBC does not support ″catalog″ as a qualifier for table name.

Restrictions

None.

Example

Figure 9 on page 114 shows an application that prints a list of column privileges for a table.

```
/* ... */
SQLRETURN
list_column_privileges(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */

    rc = SQLColumnPrivileges(hstmt, NULL, 0, schema, SQL_NTS,
                             tablename, SQL_NTS, columnname.s, SQL_NTS);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) columnname.s, 129,
                    &columnname.ind);

    rc = SQLBindCol(hstmt, 5, SQL_C_CHAR, (SQLPOINTER) grantor.s, 129,
                    &grantor.ind);

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR, (SQLPOINTER) grantee.s, 129,
                    &grantee.ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) privilege.s, 129,
                    &privilege.ind);

    rc = SQLBindCol(hstmt, 8, SQL_C_CHAR, (SQLPOINTER) is_grantable.s, 4,
                    &is_grantable.ind);


    printf("Column Privileges for %s.%s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        sprintf(cur_name, " Column: %s\n",  columnname.s);
        if (strcmp(cur_name, pre_name) != 0) {
            printf("\n%s\n", cur_name);
            printf("   Grantor         Grantee         Privilege  Grantable\n");
            printf("   --------------- --------------- ---------- ---\n");
        }
        strcpy(pre_name, cur_name);
        printf("   %-15s", grantor.s);
        printf(" %-15s", grantee.s);
        printf(" %-10s", privilege.s);
        printf(" %-3s\n", is_grantable.s);
    }                            /* endwhile */
/* ... */
```

Figure 9. An application that retrieves user privileges on table columns

Related functions

The following functions relate to SQLColumnPrivileges() calls. Refer to the descriptions of these functions for more information about how you can use SQLColumnPrivileges() in your applications.

- “SQLColumns() - Get column information” on page 115
- “SQLTables() - Get table information” on page 391

SQLColumns() - Get column information

Purpose

Table 44. SQLColumns() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	No

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 fetch a result set that a query generates.

Syntax

```
SQLRETURN   SQLColumns       (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR     FAR   *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR     FAR   *szTableName,
                              SQLSMALLINT       cbTableName,
                              SQLCHAR     FAR   *szColumnName,
                              SQLSMALLINT       cbColumnName);
```

Function arguments

Table 45 lists the data type, use, and description for each argument in this function.

Table 45. SQLColumns() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Identifies the statement handle.
SQLCHAR *	*szCatalogName*	input	Identifies the buffer that can contain a *pattern-value* to qualify the result set. *Catalog* is the first part of a three-part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	*cbCatalogName*	input	Specifies the length, in bytes, of *szCatalogName*. This must be set to 0.
SQLCHAR *	*szSchemaName*	input	Identifies the buffer that can contain a *pattern-value* to qualify the result set by schema name.
SQLSMALLINT	*cbSchemaName*	input	Specifies the length, in bytes, of *szSchemaName*.
SQLCHAR *	*szTableName*	input	Identifies the buffer that can contain a pattern-value to qualify the result set by table name.
SQLSMALLINT	*cbTableName*	input	Specifies the length, in bytes, of *szTableName*.
SQLCHAR *	*szColumnName*	input	Identifies the buffer that can contain a pattern-value to qualify the result set by column name.
SQLSMALLINT	*cbColumnName*	input	Specifies the length, in bytes, of *szColumnName*.

Usage

This function retrieves information about the columns of a table or a set of tables. Typically, you call this function after you call SQLTables() to determine the columns of a table. Use the character strings that are returned in the TABLE_SCHEM 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. Table 46 lists the columns in the result set.

The *szSchemaName, szTableName*, and *szColumnName* arguments accept search patterns. For more information about valid search patterns, see "Input arguments on catalog functions" on page 408.

Because calls to SQLColumns() frequently result in a complex and expensive query to the catalog, use these calls sparingly, and save the results rather than repeat the calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 bytes (which is consistent with SQL92 limits). Because DB2 names are less than 128 characters, the application can choose to always set aside 128 characters (plus the nul-terminator) for the output buffer. You can alternatively call SQLGetInfo() with the *InfoType* argument set to each of the following values:
- SQL_MAX_CATALOG_NAME_LEN, to determine the length of TABLE_CAT columns that the connected DBMS supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected DBMS supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected DBMS supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected DBMS supports

Although new columns might be added and the names of the existing columns might change in future releases, the position of the current columns will remain unchanged. Table 46 lists the columns in the result set that SQLColumns() currently returns.

Table 46. Columns returned by SQLColumns()

Column number	Column name	Data type	Description
1	TABLE_CAT	VARCHAR(128)	Always null.
2	TABLE_SCHEM	VARCHAR(128)	Identifies the name of the schema that contains TABLE_NAME.
3	TABLE_NAME	VARCHAR(128) NOT NULL	Identifies the name of the table, view, alias, or synonym.
4	COLUMN_NAME	VARCHAR(128) NOT NULL	Identifies the column that is described. This column contains the name of the column of the specified table, view, alias, or synonym.
5	DATA_TYPE	SMALLINT NOT NULL	Identifies the SQL data type of the column that COLUMN_NAME indicates. This is one of the values in the Symbolic SQL Data Type column in Table 4 on page 25.
6	TYPE_NAME	VARCHAR(128) NOT NULL	Identifies the character string that represents the name of the data type that corresponds to the DATA_TYPE result set column.

Table 46. Columns returned by SQLColumns() (continued)

Column number	Column name	Data type	Description
7	COLUMN_SIZE	INTEGER	If the DATA_TYPE column value denotes a character or binary string, this column contains the maximum length in characters for the column. For date, time, timestamp data types, this is the total number of characters that are required to display the value when it is converted to character. For numeric data types, this is either the total number of digits, or the total number of bits that are allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set. See also, Table 234 on page 509.
8	BUFFER_LENGTH	INTEGER	Indicates the maximum number of bytes for the associated C buffer to store data from this column if SQL_C_DEFAULT is specified on the SQLBindCol(), SQLGetData(), and SQLBindParameter() calls. This length does not include any nul-terminator. For exact numeric data types, the length accounts for the decimal and the sign. See also Table 236 on page 511.
9	DECIMAL_DIGITS	SMALLINT	Indicates the scale of the column. NULL is returned for data types where scale is not applicable. See also Table 235 on page 510.
10	NUM_PREC_RADIX	SMALLINT	Specifies 10, 2, or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, and the COLUMN_SIZE column contains the number of bits that are 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 that are allowed for the column. For numeric data types, the DBMS can return a NUM_PREC_RADIX value of either 10 or 2. NULL is returned for data types where the NUM_PREC_RADIX column does not apply.
11	NULLABLE	SMALLINT NOT NULL	Contains SQL_NO_NULLS if the column does not accept null values. Contains SQL_NULLABLE if the column accepts null values.
12	REMARKS	VARCHAR(762)	Contains any descriptive information about the column.

Table 46. Columns returned by SQLColumns() (continued)

Column number	Column name	Data type	Description
13	COLUMN_DEF	VARCHAR(254)	Identifies the default value for the column. If the default value is a numeric literal, this column contains the character representation of the numeric literal with no enclosing single quotes. If the default value is a character string, this column is that string, enclosed in single quotes. If the default value is a *pseudo-literal*, such as for DATE, TIME, and TIMESTAMP columns, this column contains the keyword of the pseudo-literal (for example, CURRENT DATE) with no enclosing quotes. If NULL was specified as the default value, this column returns the word NULL, with no enclosing single quotes. If the default value cannot be represented without truncation, this column contains the value TRUNCATED with no enclosing single quotes. If no default value was specified, this column is null.
14	SQL_DATA_TYPE	SMALLINT NOT NULL	Indicates the SQL data type. This column is the same as the DATA_TYPE column. For datetime data types, the SQL_DATA_TYPE field in the result set is SQL_DATETIME, and the SQL_DATETIME_SUB field returns the subcode for the specific datetime data type (SQL_CODE_DATE, SQL_CODE_TIME, or SQL_CODE_TIMESTAMP).
15	SQL_DATATIME_SUB	SMALLINT	The subtype code for datetime data types can be one of the following values: • SQL_CODE_DATE • SQL_CODE_TIME • SQL_CODE_TIMESTAMP For all other data types, this column returns NULL.
16	CHAR_OCTET_LENGTH	INTEGER	Contains the maximum length in bytes for a character data type column. For single-byte character sets, this is the same as COLUMN_SIZE. For non-character data types, it is null.
17	ORDINAL_POSITION	INTEGER NOT NULL	The ordinal position of the column in the table. The first column in the table is number 1.
18	IS_NULLABLE	VARCHAR(254)	Contains the string 'NO' if the column is known to be not nullable; and 'YES' otherwise.

The result set that Table 46 on page 116 describes is identical to the X/Open CLI Columns() result set specification, which is an extended version of the SQLColumns() result set that ODBC 2.0 specifies. The ODBC SQLColumns() result set includes every column in the same position up to the REMARKS column.

DB2 ODBC applications that issue SQLColumns() against a DB2 UDB for z/OS server, Version 5 or later, should expect the result set columns that are listed in Table 46 on page 116.

Return codes

After you call SQLColumns(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 47 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 47. SQLColumns() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is open on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length argument is less than 0 and not equal to SQL_NTS.
HYC00	Driver not capable.	DB2 ODBC does not support ″catalog″ as a qualifier for table name.

Restrictions

None.

Example

Figure 10 on page 120 shows an application that queries the system catalog for information about columns in a table.

```
/* ... */
SQLRETURN
list_columns(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */

    rc = SQLColumns(hstmt, NULL, 0, schema, SQL_NTS,
                    tablename, SQL_NTS, "%", SQL_NTS);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) column_name.s, 129,
                    &column_name.ind);

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR, (SQLPOINTER) type_name.s, 129,
                    &type_name.ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_LONG, (SQLPOINTER) &length,
                    sizeof(length), &length_ind);

    rc = SQLBindCol(hstmt, 9, SQL_C_SHORT, (SQLPOINTER) &scale,
                    sizeof(scale), &scale_ind);

    rc = SQLBindCol(hstmt, 12, SQL_C_CHAR, (SQLPOINTER) remarks.s, 129,
                    &remarks.ind);

    rc = SQLBindCol(hstmt, 11, SQL_C_SHORT, (SQLPOINTER) & nullable,
                    sizeof(nullable), &nullable_ind);

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 */
/* ... */
```

Figure 10. An application that returns information about table columns

Related functions

The following functions relate to SQLColumns() calls. Refer to the descriptions of these functions for more information about how you can use SQLColumns() in your applications.

- "SQLColumnPrivileges() - Get column privileges" on page 110
- "SQLSpecialColumns() - Get special (row identifier) columns" on page 376
- "SQLTables() - Get table information" on page 391

SQLConnect() - Connect to a data source

Purpose

Table 48. SQLConnect() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLConnect() establishes a connection to the target database. The application must supply a target SQL database.

You must use SQLAllocHandle() to allocate a connection handle before you call SQLConnect().

You must call SQLConnect() before you allocate a statement handle.

Syntax

```
SQLRETURN   SQLConnect      (SQLHDBC          hdbc,
                             SQLCHAR     FAR  *szDSN,
                             SQLSMALLINT      cbDSN,
                             SQLCHAR     FAR  *szUID,
                             SQLSMALLINT      cbUID,
                             SQLCHAR     FAR  *szAuthStr,
                             SQLSMALLINT      cbAuthStr);
```

Function arguments

Table 49 lists the data type, use, and description for each argument in this function.

Table 49. SQLConnect() arguments

Data type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Specifies the connection handle for the connection.
SQLCHAR *	*szDSN*	input	Specifies the data source: the name or alias name of the database to which you are connecting.
SQLSMALLINT	*cbDSN*	input	Specifies the length , in bytes, of the contents of the *szDSN* argument.
SQLCHAR *	*szUID*	input	Specifies an authorization name (user identifier). This parameter is validated and authenticated.
SQLSMALLINT	*cbUID*	input	Specifies the length, in bytes, of the contents of the *szUID* argument. This parameter is validated and authenticated.
SQLCHAR *	*szAuthStr*	input	Specifies an authentication string (password). This parameter is validated and authenticated.
SQLSMALLINT	*cbAuthStr*	input	Specifies the length, in bytes, of the contents of the *szAuthStr* argument. This parameter is validated and authenticated.

Usage

The target database (also known as a *data source*) for IBM RDBMSs is the location name that is defined in SYSIBM.LOCATIONS when DDF is configured in the DB2 subsystem. Call SQLDataSources() to obtain a list of databases that are available for connections.

In many applications, a local database is accessed (DDF is not being used). In these cases, the local database name is the name that was set during DB2 installation as 'DB2 LOCATION NAME' on the DSNTIPR installation panel for the DB2 subsystem. Your local DB2 administration staff can provide you with this name, or you can use a null connect.

A connection that is established by SQLConnect() recognizes externally created contexts and allows multiple connections to the same data source from different contexts.

Specifying a null connect: With a *null connect*, you connect to the default local database without supplying a database name.

For a null SQLConnect(), the default connection type is the value of the CONNECTTYPE keyword, which is specified in the common section of the initialization file. To override this default value, specify the SQL_ATTR_CONNECTTYPE attribute by using one of the following functions before you issue the null SQLConnect():

- SQLSetConnectAttr()
- SQLSetEnvAttr()

Use the *szDSN* argument for SQLConnect() as follows:

- If the *szDSN* argument pointer is null or the *cbDSN* argument value is 0, you perform a null connect.

  A null connect, like any connection, requires you to allocate both an environment handle and a connection handle before you make the connection. The reasons you might code a null connect include:

  – Your DB2 ODBC application needs to connect to the default data source. (The default data source is the DB2 subsystem that is specified by the MVSDEFAULTSSID initialization file setting.)

  – Your DB2 ODBC application is mixing embedded SQL and DB2 ODBC calls, and the application connected to a data source before invoking DB2 ODBC.

  – Your DB2 ODBC application runs as a stored procedure. DB2 ODBC applications that run as stored procedures must issue a null connect.
- If the *szDSN* argument pointer is not null and the *cbDSN* argument value is not 0, DB2 ODBC issues a CONNECT statement to the data source.

Specifying length arguments: You can set the input length arguments of SQLConnect() (*cbDSN*, *cbUID*, *cbAuthStr*) either to the actual length (in bytes) of their associated data (which does not include nul-terminating characters), or to SQL_NTS to indicate that the associated data is nul-terminated.

Authenticating a user: To authenticate a user, you must pass SQLConnect() both a user ID (which you specify in the *szUID* argument) and a password (which you specify in the *szAuthStr* argument). If you specify a null or empty user ID for the *szUID* argument, SQLConnect() ignores the *szAuthStr* argument and uses the primary authorization ID that is associated with the application for authentication. SQLConnect() does not accept the space character in either the *szUID* or *szAuthStr* arguments.

Using SQLDriverConnect(): Use the more extensible SQLDriverConnect() function to connect when the application needs to override any or all of the keyword values specified for this data source in the initialization file.

Users can specify various connection characteristics (attributes) in the section of the initialization file associated with the *szDSN* data source argument. Your application should set connection attributes with SQLSetConnectAttr(). To set additional attributes, call the extended connect function, SQLDriverConnect(). You can also perform a null connect with SQLDriverConnect().

Return codes

After you call SQLConnect(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 50 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 50. SQLConnect() SQLSTATEs

SQLSTATE	Description	Explanation
08001	Unable to connect to data source.	This SQLSTATE is returned for one or more of the following reasons: • DB2 ODBC is not able to establish a connection with the data source. • The connection request is rejected because a connection that was established with embedded SQL already exists.
08002	Connection in use.	The specified connection handle is being used to establish a connection with a data source, and that connection is still open.
08004	The application server rejected establishment of the connection.	This SQLSTATE is returned for one or more of the following reasons: • The data source rejects the establishment of the connection. • The number of connections that are specified by the MAXCONN keyword has been reached.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY024	Invalid argument value.	A nonmatching double quotation mark (″) is found in the *szDSN*, *szUID*, or *szAuthStr* arguments.
HY090	Invalid string or buffer length.	This SQLSTATE is returned for one or more of the following reasons: • The specified value for the *cbDSN* argument is less than 0 and is not equal to SQL_NTS, and the *szDSN* argument is not a null pointer. • The specified value for the *cbUID* argument is less than 0 and is not equal to SQL_NTS, and the *szUID* argument is not a null pointer. • The specified value for the *cbAuthStr* argument is less than 0 and is not equal to SQL_NTS, and the *szAuthStr* argument is not a null pointer.

Table 50. SQLConnect() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY501	Invalid data source name.	An invalid data source name is specified in the *szDSN* argument.

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

Figure 11 on page 125 shows an application that makes a connection to a data source with SQLConnect().

```
/* ... */
/* 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.
*/
extern    SQLCHAR   server[SQL_MAX_DSN_LENGTH + 1];
/********************************************************************/
SQLRETURN
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc)
{
    SQLRETURN       rc;
    SQLSMALLINT     outlen;

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

/********************************************************************/
/* DBconnect2 - Connect with connection type                        */
/* Valid connection types SQL_CONCURRENT_TRANS, SQL_COORDINATED_TRANS  */
/********************************************************************/
SQLRETURN DBconnect2(SQLHENV henv,
           SQLHDBC * hdbc, SQLINTEGER contype)
           SQLHDBC * hdbc, SQLINTEGER contype, SQLINTEGER conphase)
{
    SQLRETURN       rc;
    SQLSMALLINT     outlen;

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

Figure 11. An application that connects to a data source (Part 1 of 2)

```
    rc = SQLSetConnectAttr(hdbc[0], SQL_ATTR_CONNECTTYPE,(void*)contype,SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf(">---ERROR while setting Connect Type -------------\n");
        return (SQL_ERROR);
    }
    if (contype == SQL_COORDINATED_TRANS ) {
    rc=SQLSetConnectAttr(hdbc[0],SQL_ATTR_SYNC_POINT,(void*)conphase,
SQL_NTS);
        if (rc != SQL_SUCCESS) {
            printf(">---ERROR while setting Syncpoint Phase --------\n");
            return (SQL_ERROR);
        }
    }
    rc = SQLConnect(*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
    if (rc != 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);
}
/* ... */
```

Figure 11. An application that connects to a data source (Part 2 of 2)

Related functions

The following functions relate to SQLConnect() calls. Refer to the descriptions of these functions for more information about how you can use SQLConnect() in your applications.

- "SQLAllocHandle() - Allocate a handle" on page 72
- "SQLDataSources() - Get a list of data sources" on page 127
- "SQLDisconnect() - Disconnect from a data source" on page 140
- "SQLDriverConnect() - Use a connection string to connect to a data source" on page 142
- "SQLGetConnectOption() - Return current setting of a connect option" on page 199
- "SQLSetConnectOption() - Set connection option" on page 356

SQLDataSources() - Get a list of data sources

Purpose

Table 51. SQLDataSources() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLDataSources() returns a list of available target databases, one at a time.

Before you make a connection, you usually call SQLDataSources() to determine which databases are available.

Syntax

```
SQLRETURN   SQLDataSources   (SQLHENV          henv,
                              SQLUSMALLINT     fDirection,
                              SQLCHAR     FAR  *szDSN,
                              SQLSMALLINT      cbDSNMax,
                              SQLSMALLINT FAR  *pcbDSN,
                              SQLCHAR     FAR  *szDescription,
                              SQLSMALLINT      cbDescriptionMax,
                              SQLSMALLINT FAR  *pcbDescription);
```

Function arguments

Table 52 lists the data type, use, and description for each argument in this function.

Table 52. SQLDataSources() arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Specifies the environment handle.
SQLUSMALLINT	*fDirection*	input	Requests either the first data source name in the list or the next data source name in the list. *fDirection* can contain only the following values: • SQL_FETCH_FIRST • SQL_FETCH_NEXT
SQLCHAR *	*szDSN*	output	Specifies the pointer to the buffer that holds the retrieved data source name.
SQLSMALLINT	*cbDSNMax*	input	Specifies the maximum length, in bytes, of the buffer to which the *szDSN* argument points. This should be less than or equal to SQL_MAX_DSN_LENGTH + 1.
SQLSMALLINT *	*pcbDSN*	output	Specifies the pointer to the location where the value of the maximum number of bytes that are available to return in the *szDSN* is stored.
SQLCHAR *	*szDescription*	output	Specifies the pointer to the buffer where the description of the data source is returned. DB2 ODBC returns the comment field that is associated with the database cataloged to the DBMS. **IBM specific:** IBM RDBMSs always return a blank description that is padded to 30 bytes.
SQLSMALLINT	*cbDescriptionMax*	input	Specifies the maximum length, in bytes, of the *szDescription* buffer. **IBM specific:** DB2 UDB for z/OS always returns NULL.

Table 52. SQLDataSources() arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT *	*pcbDescription*	output	Specifies the pointer to the location where this function returns the actual number of bytes that the full description of the data source requires. **IBM specific:** DB2 UDB for z/OS always returns zero.

Usage

You can call this function any time with the *fDirection* argument set to either SQL_FETCH_FIRST or SQL_FETCH_NEXT.

If you specify SQL_FETCH_FIRST, the first database in the list is always returned.

If you specify SQL_FETCH_NEXT, the database that is returned depends on when you call SQLDataSources(). At the following points in your application, SQLDataSources() returns a different database name:
- 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 no more databases are 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.

Return codes

After you call SQLDataSources(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 53 on page 129 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 53. SQLDataSources() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	This SQLSTATE is returned for one or more of the following reasons: • The data source name that is returned in the argument *szDSN* is longer than the specified value in the *cbDSNMax* argument. The *pcbDSN* argument contains the length, in bytes, of the full data source name. • The data source name that is returned in the argument *szDescription* is longer than the value specified in the *cbDescriptionMax* argument. The *pcbDescription* argument contains the length, in bytes, of the full data source description. (SQLDataSources() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
58004	Unexpected system failure.	Unrecoverable system error.
HY000	General error.	An error occurred for which no specific SQLSTATE is defined. The error message that is returned by SQLGetDiagRec() in the *MessageText* argument describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The specified value for either the *cbDSNMax*argument or the *cbDescriptionMax* argument is less than 0.
HY103	Direction option out of range.	The *fDirection* argument is not set to SQL_FETCH_FIRST or SQL_FETCH_NEXT.

Restrictions

None.

Example

Figure 12 on page 130 shows an application that prints a list of available data sources with SQLDataSources().

```
/* ... */
/******************************************************
**    - Demonstrate SQLDataSource function
**    - List available servers
**      (error checking has been ignored for simplicity)
**
**  Functions used:
**
**    SQLAllocHandle      SQLFreeHandle
**    SQLDataSources
******************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "sqlcli1.h"

int
main()
{
    SQLRETURN       rc;
    SQLHENV         henv;
    SQLCHAR         source[SQL_MAX_DSN_LENGTH + 1], description[255];
    SQLSMALLINT     buffl, desl;

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

    /* 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);
    }

    SQLFreeHandle(SQL_HANDLE_ENV, henv);

    return (SQL_SUCCESS);
}
/* ... */
```

Figure 12. An application that lists available data sources

Related functions

No functions directly relate to SQLDataSources().

SQLDescribeCol() - Describe column attributes

Purpose

Table 54. SQLDescribeCol() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLDescribeCol() returns commonly used descriptor information (column name, type, precision, scale, nullability) about a column in a result set that a query generates.

If you need only one attribute of the descriptor information, or you need an attribute that SQLDescribeCol() does not return, use SQLColAttribute() in place of SQLDescribeCol(). See “SQLColAttribute() - Get column attributes” on page 101 for more information.

Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

Usually, you call this function (or the SQLColAttribute() function) before you bind a column to an application variable. See “SQLBindCol() - Bind a column to an application variable” on page 78 for more information about binding a column to an application variable.

Syntax

```
SQLRETURN   SQLDescribeCol   (SQLHSTMT          hstmt,
                              SQLUSMALLINT      icol,
                              SQLCHAR     FAR   *szColName,
                              SQLSMALLINT       cbColNameMax,
                              SQLSMALLINT FAR   *pcbColName,
                              SQLSMALLINT FAR   *pfSqlType,
                              SQLUINTEGER FAR   *pcbColDef,
                              SQLSMALLINT FAR   *pibScale,
                              SQLSMALLINT FAR   *pfNullable);
```

Function arguments

Table 55 lists the data type, use, and description for each argument in this function.

Table 55. SQLDescribeCol() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle.
SQLUSMALLINT	*icol*	input	Specifies the column number to be described. Columns are numbered sequentially from left to right, starting at 1.
SQLCHAR *	*szColName*	output	Specifies the pointer to the buffer that is to hold the name of the column. Set this to a null pointer if you do not need to receive the name of the column.
SQLSMALLINT	*cbColNameMax*	input	Specifies the size of the buffer to which the *szColName* argument points.
SQLSMALLINT *	*pcbColName*	output	Returns the number of bytes that the complete column name requires. Truncation of column name (*szColName*) to *cbColNameMax - 1* bytes occurs if *pcbColName* is greater than or equal to *cbColNameMax*.

Table 55. SQLDescribeCol() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT *	*pfSqlType*	output	Returns the base SQL data type of column. To determine if a distinct type is associated with the column, call SQLColAttribute() with *fDescType* set to SQL_COLUMN_DISTINCT_TYPE. See the symbolic SQL data type column of Table 4 on page 25 for the data types that are supported.
SQLUINTEGER *	*pcbColDef*	output	Returns the precision of the 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 that the column can hold.
SQLSMALLINT *	*pibScale*	output	Scale of column as defined in the database (applies only to SQL_DECIMAL, SQL_NUMERIC, and SQL_TYPE_TIMESTAMP). See Table 235 on page 510 for the scale of each of the SQL data types.
SQLSMALLINT *	*pfNullable*	output	Indicates whether null values are allowed for the column with the following values: • SQL_NO_NULLS • SQL_NULLABLE

Usage

Columns are identified by a number, are numbered sequentially from left to right starting with 1, and can be described in any order.

If a null pointer is specified for any of the pointer arguments, DB2 ODBC assumes that the information is not needed by the application, and nothing is returned.

If the column is a distinct type, SQLDescribeCol() returns only the built-in type in the *pfSqlType* argument. Call SQLColAttribute() with the *fDescType* argument set to SQL_COLUMN_DISTINCT_TYPE to obtain the distinct type.

Return codes

After you call SQLDescribeCol(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

If SQLDescribeCol() returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, you can call SQLGetDiagRec() to obtain one of the SQLSTATEs that are listed in Table 56 on page 133.

Table 56. SQLDescribeCol() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The column name that is returned in the *szColName* argument is longer than the specified value in the *cbColNameMax* argument. The argument *pcbColName* contains the length, in bytes, of the full column name. (SQLDescribeCol() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE)
07005	The statement did not return a result set.	The statement that is associated with the statement handle did not return a result set. No columns exist to describe. (Call SQLNumResultCols() first to determine if any rows are in the result set.)
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • The function is called prior to SQLPrepare() or SQLExecDirect() on the statement handle. • The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The length that is specified in the *cbColNameMax* argument is less than 1.
HYC00	Driver not capable.	DB2 ODBC does not recognize the SQL data type of column that the *icol* argument specifies.
HY002	Invalid column number.	The value that the *icol* argument specifies is less than 1, or it is greater than the number of columns in the result set.

Restrictions

The ODBC-defined data type SQL_BIGINT is not supported.

Example

Figure 13 on page 134 shows an application that uses SQLDescribeCol() to retrieve descriptor information about table columns.

```
/* ... */
/*******************************************************************
** process_stmt
** - allocates a statement handle
** - executes the statement
** - determines the type of statement
**   - if no result columns exist, 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 handle
*******************************************************************/

int
process_stmt(SQLHENV henv,
             SQLHDBC hdbc,
             SQLCHAR * sqlstr)
{
    SQLHSTMT        hstmt;
    SQLSMALLINT     nresultcols;
    SQLINTEGER      rowcount;
    SQLRETURN       rc;

    /* Allocate a statement handle */
    SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
```

Figure 13. An application that retrieves column descriptor information (Part 1 of 3)

```
    /* 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);

    /* 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 {                    /* display the result set */
        display_results(hstmt, nresultcols);
    }                           /* end determine statement type */

    rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt); /* Free statement handle */

    return (0);
}                               /* end process_stmt */

/*******************************************************************
** display_results
**  - for each column
**      - get column name
**      - bind column
**  - display column headings
**  - fetch each row
**      - if value truncated, build error message
**      - if column null, set value to "NULL"
**      - display row
**      - print truncation message
**  - free local storage
*******************************************************************/
display_results(SQLHSTMT hstmt,
                SQLSMALLINT nresultcols)
{
    SQLCHAR         colname[32];
    SQLSMALLINT     coltype;
    SQLSMALLINT     colnamelen;
    SQLSMALLINT     nullable;
    SQLINTEGER      collen[MAXCOLS];
    SQLUINTEGER     precision;
    SQLSMALLINT     scale;
    SQLINTEGER      outlen[MAXCOLS];
    SQLCHAR        *data[MAXCOLS];
    SQLCHAR         errmsg[256];
    SQLRETURN       rc;
    SQLINTEGER      i;
    SQLINTEGER      x;
    SQLINTEGER      displaysize;
```

Figure 13. An application that retrieves column descriptor information (Part 2 of 3)

```
    for (i = 0; i < nresultcols; i++) {
        SQLDescribeCol(hstmt, i + 1, colname, sizeof(colname),
                       &colnamelen, &coltype, &precision, &scale, NULL);
        collen[i] = precision; /* Note, assignment of unsigned int to signed */

        /* Get display length for column */
        SQLColAttribute(hstmt, i + 1, SQL_COLUMN_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", collen[i], collen[i], colname);

        /* Allocate memory to bind column                             */
        data[i] = (SQLCHAR *) malloc(collen[i]);

        /* Bind columns to program vars, converting all types to CHAR */
        rc = SQLBindCol(hstmt, i + 1, SQL_C_CHAR, data[i], collen[i], &outlen[i]);
    }
    printf("\n");

    /* Display result rows                                            */
    while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA_FOUND) {
        errmsg[0] = '\0';
        for (i = 0; i < nresultcols; i++) {
            /* Build a truncation message for any columns truncated */
            if (outlen[i] >= collen[i]) {
                sprintf((char *) errmsg + strlen((char *) errmsg),
                        "%ld chars truncated, col %d\n",
                        outlen[i] - collen[i] + 1, i + 1);
                sprintf((char *) errmsg + strlen((char *) errmsg),
                        "Bytes to return = %ld sixe of buffer\n",
                        outlen[i], collen[i]);
            }
            if (outlen[i] == SQL_NULL_DATA)
                printf("%-*.*s", collen[i], collen[i], "NULL");
            else
                printf("%-*.*s", collen[i], 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]);
    }

}                              /* end display_results */
/* ... */
```

Figure 13. An application that retrieves column descriptor information (Part 3 of 3)

Related functions

The following functions relate to SQLDescribeCol() calls. Refer to the descriptions of these functions for more information about how you can use SQLDescribeCol() in your applications.
- "SQLColAttribute() - Get column attributes" on page 101
- "SQLExecDirect() - Execute a statement directly" on page 154

- “SQLNumResultCols() - Get number of result columns” on page 299
- “SQLPrepare() - Prepare a statement” on page 306

SQLDescribeParam() - Describe parameter marker

Purpose

Table 57. SQLDescribeParam() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLDescribeParam() retrieves the description of a parameter marker that is associated with a prepared statement.

Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

Syntax

```
SQLRETURN  SQLDescribeParam  (SQLHSTMT          hstmt,
                              SQLUSMALLINT      ipar,
                              SQLSMALLINT FAR     *pfSqlType,
                              SQLUINTEGER FAR     *pcbColDef,
                              SQLSMALLINT FAR     *pibScale,
                              SQLSMALLINT FAR     *pfNullable);
```

Function arguments

Table 58 lists the data type, use, and description for each argument in this function.

Table 58. SQLDescribeParam() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle.
SQLUSMALLINT	*ipar*	input	Specifies the parameter marker number. (Parameters are ordered sequentially from left to right in a prepared SQL statement, starting at 1.)
SQLSMALLINT *	*pfSqlType*	output	Specifies the base SQL data type.
SQLUINTEGER *	*pcbColDef*	output	Returns the precision of the parameter marker. See Appendix D, “Data conversion,” on page 509 for more details on precision, scale, length, and display size.
SQLSMALLINT *	*pibScale*	output	Returns the scale of the parameter marker. See Appendix D, “Data conversion,” on page 509 for more details on precision, scale, length, and display size.
SQLSMALLINT *	*pfNullable*	output	Indicates whether the parameter allows null values. This argument returns one of the following values: • SQL_NO_NULLS: The parameter does not allow null values; this is the default. • SQL_NULLABLE: The parameter allows null values. • SQL_NULLABLE_UNKNOWN: The driver cannot determine whether the parameter allows null values.

Usage

For distinct types, SQLDescribeParam() returns both base data types for the input parameter.

For information about a parameter marker that is associated with the SQL CALL statement, use the SQLProcedureColumns() function.

Return codes

After you call SQLDescribeParam(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 59 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 59. SQLDescribeParam() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message that indicates an internal commit is issued on behalf of the application as part of the processing that sets the specified connection attribute.
HY000	General error.	An error occurred for which no specific SQLSTATE is defined. The error message that is returned by SQLGetDiagRec() in the argument *MessageText* describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY093	Invalid parameter number.	The specified value for the *ipar* argument is less than 1 or it is greater than the number of parameters that the associated SQL statement requires.
HYC00	Driver not capable.	The data source does not support the description of input parameters.

Restrictions

None.

Related functions

The following functions relate to SQLDescribeParam() calls. Refer to the descriptions of these functions for more information about how you can use SQLDescribeParam() in your applications.
- “SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator” on page 85
- “SQLCancel() - Cancel statement” on page 97
- “SQLExecDirect() - Execute a statement directly” on page 154
- “SQLExecute() - Execute a statement” on page 160
- “SQLPrepare() - Prepare a statement” on page 306

SQLDisconnect() - Disconnect from a data source

Purpose

Table 60. SQLDisconnect() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLDisconnect() closes the connection that is associated with the database connection handle.

Before you call SQLDisconnect(), you must call SQLEndTran() if an outstanding transaction exists on this connection.

After you call this function, either call SQLConnect() to connect to another database, or call SQLFreeHandle().

Syntax

```
SQLRETURN   SQLDisconnect    (SQLHDBC          hdbc);
```

Function arguments

Table 61 lists the data type, use, and description for each argument in this function.

Table 61. SQLDisconnect() arguments

Data type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Specifies the connection handle of the connection to close.

Usage

If you call SQLDisconnect() before you free all the statement handles associated with the connection, DB2 ODBC 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, if a problem was encountered during the cleanup processing, subsequent to the disconnect, or if an event occurred independently of the application (such as communication failure) that caused the current connection to be lost, SQLDisconnect() issues SQL_SUCCESS_WITH_INFO.

After a successful SQLDisconnect() call, you can reuse the connection handle you specified in the *hdbc* argument to make another SQLConnect() or SQLDriverConnect() request.

Return codes

After you call SQLDisconnect(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 62 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 62. SQLDisconnect() SQLSTATEs

SQLSTATE	Description	Explanation
01002	Disconnect error.	An error occurs during the disconnect. However, the disconnect succeeds. SQLDisconnect returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
08003	Connection is closed.	The specified connection in the *hdbc* argument is not open.
25000 or **25**501	Invalid transaction state.	A transaction is in process for the connection that the *hdbc* argument specifies. The transaction remains active, and the connection cannot be disconnected. This error does not apply to stored procedures that are written in DB2 ODBC.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.

Restrictions

None.

Example

See Figure 14 on page 146.

Related functions

The following functions relate to SQLDisconnect() calls. Refer to the descriptions of these functions for more information about how you can use SQLDisconnect() in your applications.
- “SQLAllocHandle() - Allocate a handle” on page 72
- “SQLConnect() - Connect to a data source” on page 121
- “SQLDriverConnect() - Use a connection string to connect to a data source” on page 142
- “SQLTransact() - Transaction management” on page 396

SQLDriverConnect() - Use a connection string to connect to a data source

Purpose

Table 63. SQLDriverConnect() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

SQLDriverConnect() is an alternative to SQLConnect(). Both functions establish a connection to the target database, but SQLDriverConnect() supports additional connection parameters.

Use SQLDriverConnect() when you want to specify any or all keyword values that are defined in the DB2 ODBC initialization file when you connect to a data source.

When a connection is established, the complete connection string is returned. Applications can store this string for future connection requests, which allows you to override any or all keyword values in the DB2 ODBC initialization file.

Syntax

Generic

```
SQLRETURN SQLDriverConnect   (SQLHDBC          hdbc,
                              SQLHWND          hwnd,
                              SQLCHAR      FAR *szConnStrIn,
                              SQLSMALLINT      cbConnStrIn,
                              SQLCHAR      FAR *szConnStrOut,
                              SQLSMALLINT      cbConnStrOutMax,
                              SQLSMALLINT  FAR *pcbConnStrOut,
                              SQLUSMALLINT     fDriverCompletion);
```

Function arguments

Table 64 lists the data type, use, and description for each argument in this function.

Table 64. SQLDriverConnect() arguments

Data type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Specifies the connection handle to use for the connection.
SQLHWND	*hwindow*	input	Always specify the value NULL. This argument is not used.
SQLCHAR *	*szConnStrIn*	input	A complete, partial, or empty (null pointer) connection string. See “Usage” on page 143 for a description and the syntax of this string.
SQLSMALLINT	*cbConnStrIn*	input	Specifies the length, in bytes, of the connection string to which the *szConnStrIn* argument points.
SQLCHAR *	*szConnStrOut*	output	Points to a buffer where the complete connection string is returned. If the connection is established successfully, this buffer contains the completed connection string. Applications should allocate at least SQL_MAX_OPTION_STRING_LENGTH bytes for this buffer.
SQLSMALLINT	*cbConnStrOutMax*	input	Specifies the maximum size, in bytes, of the buffer to which the *szConnStrOut* argument points.

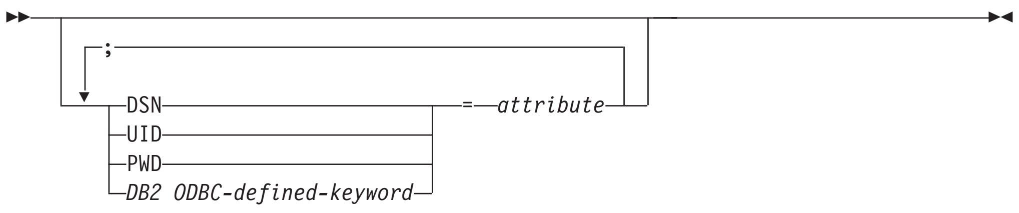
Table 64. SQLDriverConnect() arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*pcbConnStrOut*	output	Points to a buffer where the number of bytes that the complete connection string (which is returned in the *szConnStrOut* buffer) requires. If the value of *pcbConnStrOut* is greater than or equal to *cbConnStrOutMax*, the completed connection string in *szConnStrOut* is truncated to *cbConnStrOutMax* - 1 bytes.
SQLUSMALLINT	*fDriverCompletion*	input	Indicates when DB2 ODBC should prompt the user for more information. **IBM specific:** DB2 UDB for z/OS supports only the value of SQL_DRIVER_NOPROMPT for this argument. The following values are **not** supported: • SQL_DRIVER_PROMPT • SQL_DRIVER_COMPLETE • SQL_DRIVER_COMPLETE_REQUIRED

Usage

Use the connection string to pass one or more values that are needed to complete a connection. You must write the connection string to which the *szConnStrIn* argument points with the following syntax:

Connection string syntax



The connection string contains the following keywords:

DSN Data source name. The name or alias name of the database.

IBM specific: This is a required value because DB2 UDB for z/OS supports only SQL_DRIVER_NOPROMPT for the *fDriverCompletion* argument.

UID Authorization name (user identifier). This value is validated and authenticated.

IBM specific: DB2 UDB for z/OS supports only SQL_DRIVER_NOPROMPT for the *fDriverCompletion* argument. If you do not specify a value for UID, DB2 uses the primary authorization ID of your application and the PWD keyword is ignored if it is specified.

PWD The password corresponding to the authorization name. If the user ID has no password, pass an empty string (PWD=;). This value is validated and authenticated.

IBM specific: DB2 UDB for z/OS supports only SQL_DRIVER_NOPROMPT for the *fDriverCompletion* argument. The value you specify for PWD is ignored if you do not specify UID in the connection string.

The list of DB2 ODBC defined keywords and their associated attribute values are discussed in “Initialization keywords” on page 51. 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 that is associated with the first occurrence of the keyword is used.

If any keywords exist in the DB2 ODBC initialization file, the keywords and their respective values are used to augment the information that is passed to DB2 ODBC in the connection string. If the information in the DB2 ODBC initialization file contradicts information in the connection string, the values in the connection string take precedence.

The application receives an error on any value of *fDriverCompletion* as follows:

SQL_DRIVER_PROMPT:
DB2 ODBC returns SQL_ERROR.

SQL_DRIVER_COMPLETE:
DB2 ODBC returns SQL_ERROR.

SQL_DRIVER_COMPLETE_REQUIRED:
DB2 ODBC returns SQL_ERROR.

SQL_DRIVER_NOPROMPT:
The user is not prompted for any information. A connection is attempted with the information that the connection string contains. If this information is inadequate to make a connection, SQL_ERROR is returned.

When a connection is established, the complete connection string is returned.

Return codes

After you call SQLDriverConnect(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

This functions can generate all of the diagnostics listed for SQLConnect() in “Diagnostics” on page 123. Table 65 shows the additional SQLSTATEs that SQLDriverConnect() returns.

Table 65. SQLDriverConnect() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer that the*szConnstrOut* argument specifies is not large enough to hold the complete connection string. The *pcbConnStrOut* argument contains the actual length, in bytes, of the connection string that is available for return. (SQLDriverConnect() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)

Table 65. SQLDriverConnect() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
01S00	Invalid connection string attribute.	An invalid keyword or attribute value is specified in the input connection string, but the connection to the data source is successful because one of the following events occur: • The unrecognized keyword is ignored. • The invalid attribute value is ignored and the default value is used instead. (SQLDriverConnect() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
01S02	Option value changed.	SQL_CONNECTTYPE changes to SQL_CONCURRENT_TRANS while MULTICONTEXT=1 is in use.
HY090	Invalid string or buffer length.	This SQLSTATE is returned for one or more of the following reasons: • The specified value for the *cbConnStrIn* argument is less than 0 and not equal to SQL_NTS. • The specified value for the *cbConnStrOutMax* argument is less than 0.
HY110	Invalid driver completion.	The specified value for the *fDriverCompletion* argument is not equal to a valid value.

Restrictions

DB2 ODBC does not support the *hwindow* argument. Window handles do not apply in the z/OS environment.

DB2 ODBC does not support the following ODBC-defined values for the *fDriverCompletion* argument:
- SQL_DRIVER_PROMPT
- SQL_DRIVER_COMPLETE
- SQL_DRIVER_COMPLETE_REQUIRED

Example

Figure 14 on page 146 shows an application that uses SQLDriverConnect() instead of SQLConnect() to pass keyword values to the connection.

```
 /******************************************************************/
 /*  Issue SQLDriverConnect to pass a string of initialization     */
 /*  parameters to compliment the connection to the data source.   */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sqlcli1.h"

  /******************************************************************/
  /* SQLDriverConnect -----------                                   */
  /******************************************************************/

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;

   char            *ConnStrIn =
                   "dsn=STLEC1;connecttype=2;bitdata=2;optimizefornrows=30";

   char            ConnStrOut [200];
   SQLSMALLINT     cbConnStrOut;
   int             i;
   char            *token;

   (void) printf ("**** Entering CLIP10.\n\n");

  /*****************************************************************/
  /* CONNECT to DB2                                                */
  /*****************************************************************/

   rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);

   if( rc != SQL_SUCCESS )
     goto dberror;

  /*****************************************************************/
  /* Allocate connection handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocHandle( SQL_HANDLE_DBC, hEnv, &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;
```

Figure 14. An application that passes keyword values as it connects (Part 1 of 3)

```
/*****************************************************************/
/* Invoke SQLDriverConnect -----------                           */
/*****************************************************************/

 RETCODE = SQLDriverConnect (hDbc                 ,
                             NULL                 ,
                             (SQLCHAR *)ConnStrIn ,
                             strlen(ConnStrIn)    ,
                             (SQLCHAR *)ConnStrOut,
                             sizeof(ConnStrOut)   ,
                             &cbConnStrOut        ,
                             SQL_DRIVER_NOPROMPT);
 if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
 {
   (void) printf ("**** Driver Connect Failed. rc = %d.\n", RETCODE);
   goto dberror;
 }

/*****************************************************************/
/* Enumerate keywords and values returned from SQLDriverConnect  */
/*****************************************************************/

 (void) printf ("**** ConnStrOut = %s.\n", ConnStrOut);

 for (i = 1, token = strtok (ConnStrOut, ";");
      (token != NULL);
      token = strtok (NULL, ";"), i++)
   (void) printf ("**** Keyword # %d is: %s.\n", i, token);

/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/

 RETCODE = SQLDisconnect(hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;

/*****************************************************************/
/* Deallocate connection handle                                  */
/*****************************************************************/

 RETCODE = SQLFreeHandle (SQL_HANDLE_DBC, hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
```

Figure 14. An application that passes keyword values as it connects (Part 2 of 3)

```
/*****************************************************************/
  /* Disconnect from data sources in connection table            */
  /*****************************************************************/

   SQLFreeHandle(SQL_HANDLE_ENV, hEnv);     /* Free  environment handle  */

   goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP10.\n\n");

   return(RETCODE);
}
```

Figure 14. An application that passes keyword values as it connects (Part 3 of 3)

Related functions

The following functions relate to SQLDriverConnect() calls. Refer to the descriptions of these functions for more information about how you can use SQLDriverConnect() in your applications.

- "SQLAllocHandle() - Allocate a handle" on page 72
- "SQLConnect() - Connect to a data source" on page 121

SQLEndTran() - End transaction of a connection

Purpose

Table 66. SQLEndTran() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLEndTran() requests a commit or rollback operation for all active transactions on all statements that are associated with a connection. SQLEndTran() can also request that a commit or rollback operation be performed for all connections that are associated with an environment.

Syntax

```
SQLRETURN  SQLEndTran (SQLSMALLINT      HandleType,
                       SQLHANDLE        Handle,
                       SQLSMALLINT      CompletionType);
```

Function arguments

Table 67 lists the data type, use, and description for each argument in this function.

Table 67. SQLEndTran() arguments

Data type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	Identifies the handle type. Contains either SQL_HANDLE_ENV if *Handle* is an environment handle or SQL_HANDLE_DBC if *Handle* is a connection handle.
SQLHANDLE	*Handle*	input	Specifies the handle, of the type indicated by *HandleType*, that indicates the scope of the transaction. See “Usage” for more information.
SQLSMALLINT	*CompletionType*	input	Specifies whether to perform a commit or a rollback. Use one of the following values: • SQL_COMMIT • SQL_ROLLBACK

Usage

A new transaction is implicitly started when an SQL statement that can be contained within a transaction is executed against the current data source. The application might need to commit or roll back based on execution status.

If you set the *HandleType* argument to SQL_HANDLE_ENV and set the *Handle* argument to a valid environment handle, DB2 ODBC attempts to commit or roll back transactions one at a time on all connections that are in a connected state. Transactions are committed or rolled back depending on the value of the *CompletionType* argument.

If you set the *CompletionType* argument to SQL_COMMIT, SQLEndTran() issues a commit request for all statements on the connection. If *CompletionType* is SQL_ROLLBACK, SQLEndTran() issues a rollback request for all statements on the connection.

SQLEndTran() returns SQL_SUCCESS if it receives SQL_SUCCESS for each connection. If it receives SQL_ERROR on one or more connections, SQLEndTran() returns SQL_ERROR to the application, and the diagnostic information is placed in the diagnostic data structure of the environment. To determine which connections failed during the commit or rollback operation, call SQLGetDiagRec() for each connection.

Important: You must set the connection attribute SQL_ATTR_CONNECTTYPE to SQL_COORDINATED_TRANS (to indicate coordinated distributed transactions), for DB2 ODBC to provide coordinated global transactions with one-phase or two-phase commit protocols is made.

Completing a transaction has the following effects:

- Prepared SQL statements (which SQLPrepare() creates) 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 conditions are true:
  - The server is DB2 Server for VSE and VM.
  - The SQL_ATTR_CURSOR_HOLD statement attribute for this handle is set to SQL_CURSOR_HOLD_OFF.
  - The CURSORHOLD keyword in the DB2 ODBC initialization file is set so that cursor with hold is not in effect and this setting has not been overridden by resetting the SQL_ATTR_CURSOR_HOLD statement attribute.
  - The CURSORHOLD keyword is present in the SQLDriverConnect() connection string specifying that cursor-with-hold behavior is not in effect. Also you must not override this setting 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 fetch to reposition the cursor (to the next row) before continuing to process the remaining result set.

  To determine how transaction operations affect cursors, call SQLGetInfo() with the SQL_CURSOR_ROLLBACK_BEHAVIOR and SQL_CURSOR_COMMIT_BEHAVIOR attributes.
- Cursors are closed after a rollback, and all pending results are discarded.
- Statement handles are still valid after a call to SQLEndTran(), and they can be reused for subsequent SQL statements or deallocated by calling SQLFreeStmt() or SQLFreeHandle() with *HandleType* set to SQL_HANDLE_STMT.
- Cursor names, bound parameters, and column bindings survive transactions.

Regardless of whether DB2 ODBC is in autocommit mode or manual-commit mode, SQLEndTran() always sends the request to the database for execution. For more information, see “When to call SQLEndTran()” on page 21.

Return codes

After you call SQLGetDiagRec(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 68 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 68. SQLEndTran() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	An informational message was generated. (SQLEndTran() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
08003	Connection is closed.	The connection handle is not in a connected state.
08007	Connection failure during transaction.	The connection that is associated with the *Handle* argument failed during the execution of the function. No indication of whether the requested commit or rollback occurred before the failure is issued.
40001	Transaction rollback.	The transaction is rolled back due to a resource deadlock with another transaction.
HY000	General error.	An error occurred for which no specific SQLSTATE exists. The error message that is returned by SQLGetDiagRec() in the buffer that the *MessageText* argument specifies, describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the memory that is required to support the execution or completion of the function.
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() is called for the statement handle and return SQL_NEED_DATA. This function is called before data was sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY012	Invalid transaction code.	The specified value for the *CompletionType* argument was neither SQL_COMMIT nor SQL_ROLLBACK.
HY092	Option type out of range.	The specified value for the *HandleType* argument was neither SQL_HANDLE_ENV nor SQL_HANDLE_DBC.

Restrictions

SQLEndTran() cannot be used if the ODBC application is executing as a stored procedure.

Example

Refer to the sample program DSN8O3VP online in the DSN810.SDSNSAMP data set or to “DSN8O3VP sample application” on page 531.

Related functions

The following functions relate to SQLEndTran() calls. Refer to the descriptions of these functions for more information about how you can use SQLEndTran() in your applications.

- “SQLGetInfo() - Get general information” on page 234
- “SQLFreeHandle() - Free a handle” on page 190
- “SQLFreeStmt() - Free (or reset) a statement handle” on page 193

SQLError() - Retrieve error information

Purpose

Table 69. SQLError() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	Yes

In the current version of DB2 ODBC, SQLGetDiagRec() replaces SQLError(). See "SQLGetDiagRec() - Get multiple field settings of diagnostic record" on page 221 for more information.

Although DB2 ODBC supports SQLError() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLError() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

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 70 lists the data type, use, and description for each argument in this function.

Table 70. 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.

Table 70. SQLError() arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER *	*pfNativeError*	output	Native error code. In DB2 ODBC, the *pfNativeError* argument contains the SQLCODE value returned by the DBMS. If the error is generated by DB2 ODBC and not the DBMS, then this field is set to -99999.

SQLCHAR *	*szErrorMsg*	output	Pointer to buffer to contain the implementation defined message text. If the error is detected by DB2 ODBC, then the error message is prefaced by: `[DB2 UDB for z/OS][CLI Driver]`
			
			 This preface indicates that DB2 ODBC detected the error and a connection to a database has not yet been made.
			
			 The error location, ERRLOC x:y:z, keyword value is embedded in the buffer also. This is an internal error code for diagnostics.
			
			 If the error is detected during a database connection, then the error message returned from the DBMS is prefaced by: `[DB2 UDB for z/OS][CLI Driver][`*`DBMS-name`*`]`
			
			 DBMS-name is the name that is returned by SQLGetInfo() with SQL_DBMS_NAME information type.
			
			 For example,
			 `DB2`
			 `DB2/6000`
			 `Vendor`
			
			 `Vendor` indicates a non-IBM DRDA DBMS.
			
			 If the error is generated by the DBMS, the IBM-defined SQLSTATE is appended to the text string.

SQLSMALLINT	*cbErrorMsgMax*	input	The maximum (that is, the allocated) length, in bytes, 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 nul-terminator.

SQLExecDirect() - Execute a statement directly

Purpose

Table 71. SQLExecDirect() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLExecDirect() directly executes an SQL statement. SQLExecDirect() prepares and executes the SQL statement in one step.

If you plan to execute an SQL statement more than once, or if you need to obtain information about columns in the result set before you execute a query, use SQLPrepare() and SQLExecute() instead of SQLExecDirect(). For more information, see “SQLPrepare() - Prepare a statement” on page 306, and “SQLExecute() - Execute a statement” on page 160.

To use SQLExecDirect(), the connected database server must be able to dynamically prepare statement. (For more information about supported SQL statements see Table 1 on page 4.)

Syntax

```
SQLRETURN   SQLExecDirect    (SQLHSTMT          hstmt,
                               SQLCHAR     FAR   *szSqlStr,
                               SQLINTEGER        cbSqlStr);
```

Function arguments

Table 72 lists the data type, use, and description for each argument in this function.

Table 72. SQLExecDirect() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle on which you execute the SQL statement. No open cursor can be associated with the statement handle you use for this argument; see “SQLFreeStmt() - Free (or reset) a statement handle” on page 193 for more information.
SQLCHAR *	*szSqlStr*	input	Specifies the string that contains the SQL statement. The connected database server must be able to prepare the statement; see Table 1 on page 4 for more information.
SQLINTEGER	*cbSqlStr*	input	Specifies the length, in bytes, of the contents of the *szSqlStr* argument. The length must be set to either the exact length of the statement, or if the statement is nul-terminated, set to SQL_NTS.

Usage

If the SQL statement text contains vendor escape clause sequences, DB2 ODBC first modifies the SQL statement text to the appropriate DB2-specific format before submitting it for preparation and execution. If your application does not generate SQL statements that contain vendor escape clause sequences (as described in “Using vendor escape clauses” on page 465), you should set the

SQL_ATTR_NOSCAN statement attribute to SQL_NOSCAN_ON at the connection level. When you set this attribute to SQL_NOSCAN_ON, you avoid the performance impact that statement scanning causes.

The SQL statement cannot be COMMIT or ROLLBACK. Instead, You must call SQLEndTran() to issue COMMIT or ROLLBACK statements. For more information about supported SQL statements, see Table 1 on page 4.

The SQL statement string can contain parameter markers. A parameter marker is represented by a question mark (?) character, and it is used to indicate a position in the statement where an application-supplied value is to be substituted when SQLExecDirect() is called. You can obtain values for parameter markers from the following sources:

- An application variable.

  SQLBindParameter() is used to bind the application storage area to the parameter marker.
- A LOB value residing at the server that is referenced by a LOB locator.

  SQLBindParameter() 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 transferred to the application before being used as the input parameter value for another SQL statement.

You must bind all parameters before you call SQLExecDirect().

If the SQL statement is a query, SQLExecDirect() generates a cursor name and opens a cursor. If the application has used SQLSetCursorName() to associate a cursor name with the statement handle, DB2 ODBC associates the application-generated cursor name with the internally generated one.

If a result set is generated, SQLFetch() or SQLExtendedFetch() retrieves the next row or rows of data into bound variables. 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.

No open cursor can exist on the statement handle before you execute an SQL statement on that handle.

If you call SQLParamOptions() to specify that an array of input parameter values is bound to each parameter marker, you need to call SQLExecDirect() only once to process the entire array of input parameter values.

Return codes

After you call SQLExecDirect(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA
- SQL_NO_DATA_FOUND

For a description of each of these return code values, see “Function return codes” on page 23.

SQL_NEED_DATA is returned when the application requests data-at-execution parameter values. You call SQLParamData() and SQLPutData() to supply these values to SQLExecDirect().

SQL_SUCCESS is returned if the SQL statement is a searched UPDATE or searched DELETE and no rows satisfy the search condition. Use SQLRowCount() to determine the number of rows in a table that were affected by an UPDATE, INSERT, or DELETE statement that was executed on the table, or on a view of the table.

Diagnostics

Table 73 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 73. SQLExecDirect() SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	The *szSqlStr* argument contains an UPDATE or DELETE statement but no WHERE clause. (The function returns SQL_SUCCESS_WITH_INFO or SQL_NO_DATA_FOUND if no rows are in the table.)
07001	Wrong number of parameters.	The number of parameters that are bound to application variables with SQLBindParameter() is less than the number of parameter markers in the SQL statement that the *szSqlStr* argument specifies.
07006	Invalid conversion.	Transfer of data between DB2 ODBC and the application variables would result in incompatible data conversion.
21S01	Insert value list does not match column list.	The *szSqlStr* argument contains an INSERT statement and the number of values that are to be inserted do not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	The *szSqlStr* argument contains a CREATE VIEW statement, and the number of specified names is not the same degree as the derived table that is defined by the query specification.
22001	String data right truncation.	A character string that is assigned to a character type column exceeds the maximum length of the column.
22008	Invalid datetime format or datetime field overflow.	This SQLSTATE is returned for one or more of the following reasons: • The *szSqlStr* argument contains an SQL statement with an invalid datetime format. (That is, an invalid string representation or value is specified, or the value is an invalid date.) • Datetime field overflow occurred. **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.	The *szSqlStr* argument contains an SQL statement with an arithmetic expression that caused division by zero.

Table 73. SQLExecDirect() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22018	Error in assignment.	This SQLSTATE is returned for one or more of the following reasons: • The *szSqlStr* argument contains 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 that is associated with a parameter value (the contents of the *pcbValue* buffer that is specified with the SQLBindParameter() function) is not valid. • The *fSqlType* argument that is used in SQLBindParameter() 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.
23000	Integrity constraint violation.	The execution of the SQL statement is not permitted because the execution would cause an integrity constraint violation in the DBMS.
24000	Invalid cursor state.	A cursor is open on the statement handle.
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.	Results are pending on the statement handle from a previous query, or a cursor that is associated with the statement handle had not been closed.
34000	Invalid cursor name.	The *szSqlStr* argument contains a positioned DELETE or a positioned UPDATE statement, and the cursor that the statement references is not open.
37*xxx*[1]	Invalid SQL syntax.	The *szSqlStr* argument contains one or more of the following statement types: • 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 the SQL statement belongs is rolled back due to a deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
42*xxx*[1]	Syntax error or access rule violation	These SQLSTATEs indicate one of the following errors: • For **42**5*xx*, the authorization ID does not have permission to execute the SQL statement that the *szSqlStr* argument contains. • For **42***xxx*, a variety of syntax or access problems with the statement occur.
42895	The value of a host variable in the EXECUTE or OPEN statement cannot be used because of its data type	This SQLSTATE is returned for one or more of the following reasons: • The LOB locator type that is specified on the bind parameter function call does not match the LOB data type of the parameter marker. • The *fSqlType* argument, which is used on the bind parameter function, specifies a LOB locator type, but the corresponding parameter marker is not a LOB.
42S01	Database object already exists.	The *szSqlStr* argument contains a CREATE TABLE or CREATE VIEW statement, and the specified table name or view name already exists.
42S02	Database object does not exist.	The *szSqlStr* argument contains an SQL statement that references a table name or view name that does not exist.

Table 73. SQLExecDirect() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
42S11	Index already exists.	The *szSqlStr* argument contains a CREATE INDEX statement, and the specified index name already exists.
42S12	Index not found.	The *szSqlStr* argument contains a DROP INDEX statement, and the specified index name does not exist.
42S21	Column already exists.	The *szSqlStr* argument contains an ALTER TABLE statement, and the column that is specified in the ADD clause is not unique or identifies an existing column in the base table.
42S22	Column not found.	The *szSqlStr* argument contains an SQL statement that references a column name that does not exist.
44000	Integrity constraint violation.	When the *szSqlStr* argument contains an SQL statement with a parameter or literal, one of the following violations occur: • The parameter value is NULL for a column that is defined as NOT NULL in the associated table column. • A duplicate value is supplied for a column that is constrained to contain only unique values. • An integrity constraint is violated.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	The *szSqlStr* argument specifies a null pointer.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY019	Numeric value out of range.	This SQLSTATE is returned for one or more of the following reasons: • A numeric value that is 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. • The *szSqlStr* argument contains an SQL statement with an arithmetic expression that causes division by zero.
HY090	Invalid string or buffer length.	The argument *cbSqlStr* is less than 1 but not equal to SQL_NTS.

Note:

1. *xxx* refers to any SQLSTATE with that class code. For example, **37***xxx* refers to any SQLSTATE with class code '**37**'.

Restrictions

None.

Example

See Figure 16 on page 176.

Related functions

The following functions relate to SQLExecDirect() calls. Refer to the descriptions of these functions for more information about how you can use SQLExecDirect() in your applications.

- “SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator” on page 85
- “SQLExecute() - Execute a statement” on page 160
- “SQLExtendedFetch() - Fetch an array of rows” on page 163
- “SQLFetch() - Fetch the next row” on page 171
- “SQLParamData() - Get next parameter for which a data value is needed” on page 301
- “SQLPutData() - Pass a data value for a parameter” on page 335
- “SQLSetParam() - Bind a parameter marker to a buffer” on page 364

SQLExecute() - Execute a statement

Purpose

Table 74. SQLExecute() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLExecute() executes a statement, which you successfully prepared with SQLPrepare(), once or multiple times. When you execute a statement with SQLExecute(), the current value of any application variables that are bound to parameter markers in that statement are used.

Syntax

```
SQLRETURN   SQLExecute      (SQLHSTMT         hstmt);
```

Function arguments

Table 75 lists the data type, use, and description for each argument in this function.

Table 75. SQLExecute() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle. No open cursor can be associated with the statement handle; see “SQLFreeStmt() - Free (or reset) a statement handle” on page 193 for more information.

Usage

Use SQLExecute() to execute an SQL statement that you prepared with SQLPrepare(). You can include parameter markers in this SQL statement. Parameter markers are question mark characters (?) that you place in the SQL statement string. When you call SQLExecute() to execute a statement that contains parameter markers, each of these markers is replaced with the contents of a host variable.

You must use SQLBindParameter() to associate all parameter markers in the statement string to an application-supplied values before you call SQLExecute(). This value can be obtained from one of the following sources:

- An application variable.

  SQLBindParameter() is used to bind the application storage area to the parameter marker.
- A LOB value residing at the server that is referenced by a LOB locator.

  SQLBindParameter() 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 transferred to the application before being used as the input parameter value for another SQL statement.

You must bind all parameters before you call SQLExecute().

After the application processes the results from the SQLExecute() call, it can execute the statement again with new (or the same) parameter values.

A statement that is executed by SQLExecDirect() cannot be re-executed by calling SQLExecute(); you must call SQLPrepare() before executing a statement with SQLExecute().

If the prepared SQL statement is a query, SQLExecute() generates a cursor name, and opens the cursor. If the application uses SQLSetCursorName() to associate a cursor name with the statement handle, DB2 ODBC associates the application-generated cursor name with the internally generated one.

To execute a query more than once, you must close the cursor by calling SQLFreeStmt() with the*fOption* argument set to SQL_CLOSE. No open cursor can exist on the statement handle when calling SQLExecute().

If a result set is generated, SQLFetch() or SQLExtendedFetch() retrieves the next row or rows of data into bound variables or LOB locators. You can also retrieve data by calling SQLGetData() for any column that was not bound.

If the SQL statement is a positioned DELETE or a positioned UPDATE, you must position the cursor that the statement references on a row at the time SQLExecute() is called, and define the cursor on a separate statement handle under the same connection handle.

If you call SQLParamOptions() to specify that an array of input parameter values is bound to each parameter marker, you need 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), call SQLMoreResults() to advance to the next result set when processing on the current result set is complete. See “SQLMoreResults() - Check for more result sets” on page 289 for more information.

Return codes

After you call SQLExecute(), it returns one of the following values:
- 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 requests data-at-execution parameter values. You call SQLParamData() and SQLPutData() to supply these values to SQLExecute().

SQL_SUCCESS is returned if the SQL statement is a searched UPDATE or searched DELETE and no rows satisfy the search condition. Use SQLRowCount() to determine the number of rows in a table that were affected by an UPDATE, INSERT, or DELETE statement executed on the table, or on a view of the table.

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

The SQLSTATEs that SQLExecute() returns include all the SQLSTATEs that SQLExecDirect() can generate, except for **HY**009, **HY**014, and **HY**090, and with the addition of **HY**010. SQLSTATEs for SQLExecDirect() are listed in Table 73 on page 156.

Table 76 lists and describes the additional SQLSTATE that SQLExecute() can return.

Table 76. SQLExecute() SQLSTATEs

SQLSTATE	Description	Explanation
HY010	Function sequence error.	SQLExecute() is called on a statement prior to SQLPrepare().

Restrictions

None.

Example

See Figure 26 on page 309.

Related functions

The following functions relate to SQLExecute() calls. Refer to the descriptions of these functions for more information about how you can use SQLExecute() in your applications.

- "SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator" on page 85
- "SQLExecDirect() - Execute a statement directly" on page 154
- "SQLExecute() - Execute a statement" on page 160
- "SQLExtendedFetch() - Fetch an array of rows" on page 163
- "SQLFetch() - Fetch the next row" on page 171
- "SQLParamOptions() - Specify an input array for a parameter" on page 304
- "SQLPrepare() - Prepare a statement" on page 306
- "SQLSetParam() - Bind a parameter marker to a buffer" on page 364

SQLExtendedFetch() - Fetch an array of rows

Purpose

Table 77. SQLExtendedFetch() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	No	No

SQLExtendedFetch() extends the function of SQLFetch() by returning a block of data containing multiple rows (called a *row set*), in the form of an array, for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

To fetch one row of data at a time, call SQLFetch() instead of SQLExtendedFetch().

For more information about block or array retrieval, see “Retrieving a result set into an array” on page 417.

Syntax

```
SQLRETURN SQLExtendedFetch  (SQLHSTMT          hstmt,
                             SQLUSMALLINT      fFetchType,
                             SQLINTEGER        irow,
                             SQLUINTEGER  FAR  *pcrow,
                             SQLUSMALLINT FAR  *rgfRowStatus);
```

Function arguments

Table 78 lists the data type, use, and description for each argument in this function.

Table 78. SQLExtendedFetch() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle from which you retrieve an array data.
SQLUSMALLINT	*fFetchType*	input	Specifies the direction and type of fetch. DB2 ODBC supports only the fetch direction SQL_FETCH_NEXT (that is, forward-only cursor direction). The next array (row set) of data is always retrieved.
SQLINTEGER	*irow*	input	Reserved for future use. Use any integer for this argument.
SQLUINTEGER *	*pcrow*	output	Returns the number of the rows that are actually fetched. If an error occurs during processing, the *pcrow* argument points to the ordinal position of the row (in the row set) that precedes the row where the error occurred. If an error occurs retrieving the first row, the *pcrow* argument points to the value 0.

Table 78. SQLExtendedFetch() arguments (continued)

Data type	Argument	Use	Description
SQLUSMALLINT *	*rgfRowStatus*	output	Returns an array of status values. The number of elements must equal the number of rows in the row set (as defined by the SQL_ATTR_ROWSET_SIZE attribute). A status value for each row that is fetched is returned: • SQL_ROW_SUCCESS If the number of rows fetched is less than the number of elements in the status array (that is, less than the row set size), the remaining status elements are set to SQL_ROW_NOROW. DB2 ODBC cannot detect whether a row has been updated or deleted since the start of the fetch. Therefore, the following ODBC-defined status values are not reported: • SQL_ROW_DELETED • SQL_ROW_UPDATED

Usage

SQLExtendedFetch() performs 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.

You cannot mix SQLExtendedFetch() with SQLFetch() when you retrieve results.

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 row set that was just retrieved.

The number of elements in the *rgfRowStatus* array output buffer must equal the number of rows in the row set (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.

For any columns in the result set that are bound using the SQLBindCol() function, DB2 ODBC converts the data for the bound columns as necessary and stores it in the locations that are bound to these columns. As mentioned in "Retrieving a result set into an array" on page 417, the result set can be bound in a column-wise or row-wise fashion.

Binding column-wise: To bind a result set in column-wise fashion, an application specifies SQL_BIND_BY_COLUMN for the SQL_BIND_TYPE statement attribute. (This is the default value.) Then the application calls the SQLBindCol() function.

When you call 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 the number of bytes that you specify with the *cbValueMax* argument in the SQLBindCol() call. If, however, the associated C buffer type is fixed-width (such as SQL_C_LONG), the data is stored at an offset corresponding to that fixed-length from the data for the previous row.

For each bound column, the number of bytes that are available to return for each element is stored in the array buffer that the *pcbValue* argument on SQLBindCol()

specifies. The number of bytes that are available to return for the first row of that column is stored at the start of the buffer. The number of bytes available to return for each subsequent row is stored at an offset equal to the value that the following C function returns:

```
sizeof(SQLINTEGER)
```

If the data in the column is null for a particular row, the associated element in the array that the *pcbValue* argument in SQLBindCol() array is set to SQL_NULL_DATA.

Binding row-wise: The application needs to first call SQLSetStmtAttr() with the SQL_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* argument in SQLBindCol(). Each subsequent row of data is separated by an offset equal to the number of bytes that you specify in the *vParam* argument in SQLSetStmtAttr() from the data for the previous row.

For each bound column, the number of bytes that are available to return for the first row is stored at the address given by the *pcbValue* argument in SQLBindCol(). Each subsequent value is separated by an offset equal to the number of bytes you specify in the *vParam* argument in SQLBindCol().

Handling errors: If SQLExtendedFetch() returns an error that applies to the entire row set, the SQL_ERROR function return code is reported with the appropriate SQLSTATE. The contents of the row set buffer are undefined and the cursor position is unchanged.

If an error occurs that applies to a single row:

- The corresponding element in the *rgfRowStatus* array for the row is set to SQL_ROW_ERROR
- An SQLSTATE of **01**S01 is added to the list of errors that you can obtain with SQLGetDiagRec()
- Zero or more additional SQLSTATEs that describe the error for the current row are added to the list of errors that you can obtain with SQLGetDiagRec()

When the value SQL_ROW_ERROR appears in the array that the *rgfRowStatus* argument specifies, this value indicates that an error occurred with the corresponding element. This array 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 ODBC continues to fetch the remaining rows in the row set and returns SQL_SUCCESS_WITH_INFO as the function return code. Each row that encounters an error receives an SQLSTATE of **01**S01 and zero or more additional SQLSTATEs that indicate the errors in the row. Retrieve this information with SQLGetDiagRec(). Individual errors that apply to specific rows do not affect the cursor, which continues to advance.

Handling encoding schemes: The CURRENTAPPENSCH keyword in the initialization file and the *fCType* argument in SQLBindCol() or SQLGetData() determine the encoding scheme of any character or graphic data in the result set. See “CURRENTAPPENSCH” on page 53 for additional information about the CURRENTAPPENSCH keyword. See “SQLBindCol() - Bind a column to an application variable” on page 78 and “SQLGetData() - Get data from a column” on page 207 for additional information about the *fCType* argument.

Return codes

After you call SQLExtendedFetch(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 79 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 79. SQLExtendedFetch() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data that is returned for one **or more** columns is truncated. (SQLExtendedFetch() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
01S01	Error in row.	An error occurs while fetching one or more rows. (SQLExtendedFetch() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
07002	Too many columns.	This SQLSTATE is returned for one or more of the following reasons: • A column number that is specified in the bind of one or more columns is greater than the number of columns that are in the result set. • The application uses SQLSetColAttributes() to inform DB2 ODBC of the descriptor information of the result set, but it does not provide this information for every column that is in the result set.
07006	Invalid conversion.	The data value can not be converted in a meaningful manner to the data type that the *fCType* argument in SQLBindCol() specifies.
22002	Invalid output or indicator buffer specified.	The *pcbValue* argument in SQLBindCol() specifies a null pointer and the value of the corresponding column is null. The function can not report SQL_NULL_DATA.
22008	Invalid datetime format or datetime field overflow.	This SQLSTATE is returned for one or more of the following reasons: • Conversion from character string to datetime format is indicated, but an invalid string representation or value is specified, or the value is an invalid date. • The value of a date, time, or timestamp does not conform to the syntax for the data type that is specified. • Datetime field overflow occurred. **Example:** An arithmetic operation on a date or timestamp produces 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 is returned that results in division by zero.

Table 79. SQLExtendedFetch() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22018	Error in assignment.	This SQLSTATE is returned for one or more of the following reasons: • A returned value is incompatible with the data type of the bound column. • A returned LOB locator was incompatible with the data type of the bound column.
24000	Invalid cursor state.	The SQL statement that is executed on the statement handle is not a query.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • SQLExtendedFetch() is called on a statement handle after a SQLFetch() call, and before the SQLFreeStmt() (with the *fOption* argument set to SQL_CLOSE) call. • The function is called prior to calling SQLPrepare() or SQLExecDirect() on the statement handle. • The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY019	Numeric value out of range.	This SQLSTATE is returned for one or more of the following reasons: • A numeric value (as numeric or string) that is returned for one or more columns causes the whole part of a number to be truncated either at the time of assignment or in computing an intermediate result. • A value from an arithmetic expression is returned that results in division by zero.
HY106	Fetch type out of range.	The value that the*fFetchType* argument specifies is not recognized.
HYC00	Driver not capable.	This SQLSTATE is returned for one or more of the following reasons: • DB2 ODBC or the data source does not support the conversion that the *fCType*argument in SQLBindCol() and the SQL data type of the corresponding column require. • A call to SQLBindCol() is made for a column data type that DB2 ODBC does not support. • The specified fetch type is recognized, but it is not supported.

Restrictions

Although this function is deprecated in ODBC 3.0, this function is not deprecated in DB2 ODBC. DB2 ODBC does not support SQLFetchScroll(), which replaces SQLExtendedFetch() in ODBC 3.0.

Example

Figure 15 shows an application that uses SQLExtendedFetch() to perform an array fetch.

```
/* ... */
    "SELECT deptnumb, deptname, id, name FROM staff, org \
                     WHERE dept=deptnumb AND job = 'Mgr'";

    /* Column-wise */
    SQLINTEGER      deptnumb[ROWSET_SIZE];

    SQLCHAR         deptname[ROWSET_SIZE][15];
    SQLINTEGER      deptname_l[ROWSET_SIZE];

    SQLSMALLINT     id[ROWSET_SIZE];

    SQLCHAR         name[ROWSET_SIZE][10];
    SQLINTEGER      name_l[ROWSET_SIZE];
    /* Row-wise (Includes buffer for both column data and length) */
    struct {
        SQLINTEGER      deptnumb_l; /* length */
        SQLINTEGER      deptnumb; /* value  */
        SQLINTEGER      deptname_l;
        SQLCHAR         deptname[15];
        SQLINTEGER      id_l;
        SQLSMALLINT     id;
        SQLINTEGER      name_l;
        SQLCHAR         name[10];
    }               R[ROWSET_SIZE];

    SQLUSMALLINT    Row_Stat[ROWSET_SIZE];
    SQLUINTEGER     pcrow;
    int             i;
/* ... */
```

Figure 15. An application that performs an array fetch (Part 1 of 3)

```
/*********************************************/
/* Column-wise binding                       */
/*********************************************/
rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);

rc = SQLSetStmtAttr(hstmt, SQL_ATTR_ROWSET_SIZE, (void*) ROWSET_SIZE, 0);

rc = SQLExecDirect(hstmt, stmt, SQL_NTS);

rc = SQLBindCol(hstmt, 1, SQL_C_LONG, (SQLPOINTER) deptnumb, 0, NULL);

rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) deptname,
                15, deptname_l);

rc = SQLBindCol(hstmt, 3, SQL_C_SSHORT, (SQLPOINTER) id, 0, NULL);

rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) name, 10, name_l);

/* Fetch ROWSET_SIZE rows ast a time, and display */
printf("\nDEPTNUMB DEPTNAME         ID       NAME\n");
printf("-------- -------------- -------- ---------\n");
while ((rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0,
        &pcrow, Row_Stat)) == SQL_SUCCESS) {
    for (i = 0; i < pcrow; i++) {
        printf("%8ld %-14s %8ld %-9s\n", deptnumb[i], deptname[i],
               id[i], name[i]);
    }
    if (pcrow < ROWSET_SIZE)
        break;
}                            /* endwhile */

if (rc != SQL_NO_DATA_FOUND && rc != SQL_SUCCESS)
    CHECK_HANDLE(SQL_HANDLE_STMT, hstmt, rc);

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

Figure 15. An application that performs an array fetch (Part 2 of 3)

```
    /********************************************/
    /* Row-wise binding              */
    /********************************************/
    rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
    CHECK_HANDLE(SQL_HANDLE_STMT, hstmt, rc);

    /* Set maximum number of rows to receive with each extended fetch */
    rc = SQLSetStmtAttr(hstmt, SQL_ATTR_ROWSET_SIZE, (void*) ROWSET_SIZE, 0);
    CHECK_HANDLE(SQL_HANDLE_STMT, hstmt, rc);

    /*
     * Set vparam to size of one row, used as offset for each bindcol
     * rgbValue
     */
    /* ie. &(R[0].deptnumb) + vparam = &(R[1].deptnum) */
    rc = SQLSetStmtAttr(hstmt, SQL_ATTR_BIND_TYPE,
                     (void*) (sizeof(R) / ROWSET_SIZE), 0);

    rc = SQLExecDirect(hstmt, stmt, SQL_NTS);

    rc = SQLBindCol(hstmt, 1, SQL_C_LONG, (SQLPOINTER) &R[0].deptnumb, 0,
                &R[0].deptnumb_l);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) R[0].deptname, 15,
                &R[0].deptname_l);

    rc = SQLBindCol(hstmt, 3, SQL_C_SSHORT, (SQLPOINTER) &R[0].id, 0,
                &R[0].id_l);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) R[0].name, 10, &R[0].name_l);

    /* Fetch ROWSET_SIZE rows at a time, and display */
    printf("\nDEPTNUMB DEPTNAME         ID       NAME\n");
    printf("-------- -------------- -------- ---------\n");
    while ((rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, Row_Stat))
           == SQL_SUCCESS) {
        for (i = 0; i < pcrow; i++) {
            printf("%8ld %-14s %8ld %-9s\n", R[i].deptnumb, R[i].deptname,
                   R[i].id, R[i].name);
        }
        if (pcrow < ROWSET_SIZE)
            break;
    }                           /* endwhile */

    if (rc != SQL_NO_DATA_FOUND && rc != SQL_SUCCESS)
        CHECK_HANDLE(SQL_HANDLE_STMT, hstmt, rc);

    /* Free handles, commit, exit */
/* ... */
```

Figure 15. An application that performs an array fetch (Part 3 of 3)

Related functions

The following functions relate to SQLExtendedFetch() calls. Refer to the descriptions of these functions for more information about how you can use SQLExtendedFetch() in your applications.

- “SQLExecute() - Execute a statement” on page 160
- “SQLExecDirect() - Execute a statement directly” on page 154
- “SQLFetch() - Fetch the next row” on page 171

SQLFetch() - Fetch the next row

Purpose

Table 80. SQLFetch() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLFetch() advances the cursor to the next row of the result set, and retrieves any bound columns.

Columns can be bound to the following locations:
- Application storage
- LOB locators

When you call SQLFetch(), DB2 ODBC performs the appropriate data transfer, along with any data conversion that was indicated when you bound the column. You can call SQLGetData() to retrieve the columns individually after the fetch.

You can call SQLFetch() only after you generate a result set. Any of the following actions generate a result set:
- Executing a query
- Calling SQLGetTypeInfo()
- Calling a catalog function

To retrieve multiple rows at a time, use SQLExtendedFetch().

Syntax

```
SQLRETURN   SQLFetch        (SQLHSTMT         hstmt);
```

Function arguments

Table 81 lists the data type, use, and description for each argument in this function.

Table 81. SQLFetch() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle from which to fetch data.

Usage

Call SQLFetch() to retrieve results into bound application variables and to advance the position of the cursor in a result set. You can call SQLFetch() only after a result set is generated on the statement handle. Before you call SQLFetch() for 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() fails.

When you retrieve all the rows from the result set, or do not need the remaining rows, call SQLFreeStmt() or SQLCloseCursor() to close the cursor and discard the remaining data and associated resources.

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() can be called to obtain all of the columns individually. Data in unbound columns is discarded when SQLFetch() advances the cursor to the next row. For fixed-length data types, or small varying-length data types, binding columns provides better performance than using SQLGetData().

Columns can be bound to application storage or you can use LOB locators.

Fetching into application storage: SQLBindCol() binds application storage to the column. You transfer data from the server to the application when you call SQLFetch(). The length of the data that is available to return is also set.

If LOB values are too large to retrieve in one fetch, retrieve these values in pieces either by using SQLGetData() (which can be used for any column type), or by binding a LOB locator and using SQLGetSubString().

Fetching into LOB locators: SQLBindCol() is used to bind LOB locators to the column. Only the LOB locator (4 bytes) is transferred from the server to the application at fetch time.

When your application receives a locator, it can use the locator 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 are created (even when the cursor moves to another row), or until they are freed using the FREE LOCATOR statement.

Handling data truncation: If any bound storage buffers are not large enough to hold the data returned by SQLFetch(), the data is 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* contains the actual length, in bytes, 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 are 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 (see “Diagnostics” on page 173).

Truncation of graphic data types is treated the same as character data types, except that the buffer you specify in the *rgbValue* argument for SQLBindCol(). This buffer is filled to the nearest multiple of two bytes that is less than or equal to the value you specify in the *cbValueMax* argument for SQLBindCol(). Graphic (DBCS) data transferred between DB2 ODBC and the application is not nul-terminated if the C buffer type is SQL_C_CHAR. If the buffer type is SQL_C_DBCHAR, then nul-termination of graphic data does occur.

To eliminate warnings when data is truncated, call SQLSetStmtAttr() with the SQL_ATTR_MAX_LENGTH attribute set to a maximum length value. Then allocate a buffer for the *rgbValue* argument that is the same number of bytes (plus the nul-terminator) as the value you specified for SQL_ATTR_MAX_LENGTH. If the column data is larger than the maximum length that you specified for

SQL_ATTR_MAX_LENGTH, SQL_SUCCESS is returned. When you specify a maximum length, the length you specify, not the actual length, is returned in the *pcbValue* argument.

To retrieve multiple rows at a time, use SQLExtendedFetch(). You cannot mix SQLFetch() calls with SQLExtendedFetch() calls on the same statement handle.

Return codes

After you call SQLFetch(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND is returned if no rows are in the result set, or previous SQLFetch() calls have fetched all the rows from the result set. If all the rows are fetched, the cursor is positioned after the end of the result set.

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 82 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 82. SQLFetch() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data that is returned for one or more columns is truncated. String values or numeric values are truncated on the right. (SQLFetch() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
07002	Too many columns.	This SQLSTATE is returned for one or more of the following reasons: • A column number that is specified in the bind for one or more columns is greater than the number of columns in the result set. • The application uses SQLSetColAttributes() to inform DB2 ODBC of the descriptor information of the result set, but does not provide this information for every column in the result set.
07006	Invalid conversion.	The data value cannot be converted in a meaningful manner to the data type that the *fCType* argument in SQLBindCol() specifies.
22002	Invalid output or indicator buffer specified.	The *pcbValue* argument in SQLBindCol() specifies a null pointer, and the value of the corresponding column is null. The function can not report SQL_NULL_DATA.

Table 82. SQLFetch() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22008	Invalid datetime format or datetime field overflow.	This SQLSTATE is returned for one or more of the following reasons: • Conversion from character string to datetime format is indicated, but an invalid string representation or value is specified, or the value is an invalid date. • The value of a date, time, or timestamp does not conform to the syntax for the specified data type. • Datetime field overflow occurred. **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 is returned that results in division by zero.
22018	Error in assignment.	This SQLSTATE is returned for one or more of the following reasons: • A returned value is incompatible with the data type of binding. • A returned LOB locator is incompatible with the data type of the bound column.
24000	Invalid cursor state.	The previous SQL statement that is executed on the statement handle is not a query.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
54028	Maximum LOB locator assigned.	The maximum number of concurrent LOB locators has been reached. A new locator can not be assigned.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
| **HY**002	Invalid column number.	This SQLSTATE is returned for one or more of the following reasons: • The specified column is less than 0 or greater than the number of result columns. • The specified column is 0, but DB2 ODBC does not support ODBC bookmarks (*icol* = 0). • SQLExtendedFetch() is called for this result set.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • SQLFetch() is called for a statement handle after SQLExtendedFetch() and before SQLCloseCursor(). • The function is called prior to SQLPrepare() or SQLExecDirect(). • The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.

Table 82. SQLFetch() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY019	Numeric value out of range.	This SQLSTATE is returned for one or more of the following reasons: • Returning the numeric value (as numeric or string) for one or more columns causes 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 is returned that results in division by zero. **Important:** The associated cursor is undefined if this error is detected by DB2 UDB for z/OS. If the error is detected by DB2 UDB or by other IBM RDBMSs, the cursor remains open and continues to advance on subsequent fetch calls.
HYC00	Driver not capable.	This SQLSTATE is returned for one or more of the following reasons: • DB2 ODBC or the data source does not support the conversion that the *fCType* argument in SQLBindCol() and the SQL data type of the corresponding column require. • A call to SQLBindCol() was made for a column data type that is not supported by DB2 ODBC.

Restrictions

None.

Example

Figure 16 on page 176 shows an application that uses SQLFetch() to retrieve data from bound columns of a result set.

```
/* ... */
/******************************************************************
** main
*******************************************************************/
int
main( int argc, char * argv[] )
{
    SQLHENV         henv;
    SQLHDBC         hdbc;
    SQLHSTMT        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));
    rc = DBconnect(henv, &hdbc);/* allocate a connect handle, and connect */
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));
    rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
    rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);
    rc = SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) deptname.s, 15,
                    &deptname.ind);
    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) location.s, 15,
                    &location.ind);

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, RETCODE);

    rc = SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
    rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);

    printf("Disconnecting .....\n");

    rc = SQLDisconnect(hdbc);
    rc = SQLFreeHandle (SQL_HANDLE_DBC, hdbc);
    rc = SQLFreeHandle (SQL_HANDLE_DBC, henv);
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));
}                               /* end main */
/* ... */
```

Figure 16. An application that retrieves data from bound columns

Related functions

The following functions relate to SQLFetch() calls. Refer to the descriptions of these functions for more information about how you can use SQLFetch() in your applications.

- "SQLExtendedFetch() - Fetch an array of rows" on page 163
- "SQLExecute() - Execute a statement" on page 160
- "SQLExecDirect() - Execute a statement directly" on page 154
- "SQLGetData() - Get data from a column" on page 207

SQLForeignKeys() - Get a list of foreign key columns

Purpose

Table 83. SQLForeignKeys() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

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          hstmt,
                             SQLCHAR      FAR  *szPkCatalogName,
                             SQLSMALLINT       cbPkCatalogName,
                             SQLCHAR      FAR  *szPkSchemaName,
                             SQLSMALLINT       cbPkSchemaName,
                             SQLCHAR      FAR  *szPkTableName,
                             SQLSMALLINT       cbPkTableName,
                             SQLCHAR      FAR  *szFkCatalogName,
                             SQLSMALLINT       cbFkCatalogName,
                             SQLCHAR      FAR  *szFkSchemaName,
                             SQLSMALLINT       cbFkSchemaName,
                             SQLCHAR      FAR  *szFkTableName,
                             SQLSMALLINT       cbFkTableName);
```

Function arguments

Table 84 lists the data type, use, and description for each argument in this function.

Table 84. SQLForeignKeys() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle on which to return results.
SQLCHAR *	*szPkCatalogName*	input	Specifies the catalog qualifier of the primary key table. This must be a null pointer or a zero length string.
SQLSMALLINT	*cbPkCatalogName*	input	Specifies the length, in bytes, of the *szPkCatalogName* argument. This must be set to 0.
SQLCHAR *	*szPkSchemaName*	input	Specifies the schema qualifier of the primary key table.
SQLSMALLINT	*cbPkSchemaName*	input	Specifies the length, in bytes, of the *szPkSchemaName* argument.
SQLCHAR *	*szPkTableName*	input	Specifies the name of the table that contains the primary key.
SQLSMALLINT	*cbPkTableName*	input	Specifies the length, in bytes, of the *szPkTableName* argument.
SQLCHAR *	*szFkCatalogName*	input	Specifies the catalog qualifier of the table that contains the foreign key. This must be a null pointer or a zero length string.
SQLSMALLINT	*cbFkCatalogName*	input	Specifies the length, in bytes, of the *szFkCatalogName* argument. This must be set to 0.
SQLCHAR *	*szFkSchemaName*	input	Specifies the schema qualifier of the table that contains the foreign key.
SQLSMALLINT	*cbFkSchemaName*	input	Specifies the length, in bytes, of the *szFkSchemaName* argument.

Table 84. SQLForeignKeys() arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*szFkTableName*	input	Specifies the name of the table that contains the foreign key.
SQLSMALLINT	*cbFkTableName*	input	Specifies the length, in bytes, of the *szFkTableName* argument.

Usage

If the *szPkTableName* argument contains a table name and the *szFkTableName* argument 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 the *szFkTableName* argument contains a table name and the *szPkTableName* argument is an empty string, SQLForeignKeys() returns a result set that contains all of the foreign keys in the table that you specify in the *szFkTableName* argument and the all the primary keys (on other tables) to which they refer.

If both of the *szPkTableName* argument and the *szFkTableName* argument contain table names, SQLForeignKeys() returns foreign keys that refer to the primary key of the table that you specify in the *szPkTableName* argument from the table that you specify in the *szFkTableName* argument. All foreign keys that this type of SQLForeignKeys() call returns refer to a single primary key.

If you do not specify a schema qualifier argument that is associated with a table name, DB2 ODBC uses the schema name that is currently in effect for the current connection.

Table 85 lists each column in the result set that SQLForeignKeys() currently returns.

Table 85. Columns returned by SQLForeignKeys()

Column number	Column name	Data type	Description
1	PKTABLE_CAT	VARCHAR(128)	This is always NULL.
2	PKTABLE_SCHEM	VARCHAR(128)	Contains the name of the schema to which the table in PKTABLE_NAME belongs.
3	PKTABLE_NAME	VARCHAR(128) NOT NULL	Contains the name of the table on which the primary key is defined.
4	PKCOLUMN_NAME	VARCHAR(128) NOT NULL	Contains the name of the column on which the primary key is defined.
5	FKTABLE_CAT	VARCHAR(128)	This is always NULL.
6	FKTABLE_SCHEM	VARCHAR(128)	Contains the name of the schema to which the table in FKTABLE_NAME belongs.
7	FKTABLE_NAME	VARCHAR(128) NOT NULL	Contains the name of the table that on which the foreign key is defined.
8	FKCOLUMN_NAME	VARCHAR(128) NOT NULL	Contains the name of the column on which the foreign key is defined.
9	KEY_SEQ	SMALLINT NOT NULL	Contains the ordinal position of the column in the key. The first position is 1.

Table 85. Columns returned by SQLForeignKeys() (continued)

Column number	Column name	Data type	Description
10	UPDATE_RULE	SMALLINT	Identifies the action that is applied to the foreign key when the SQL operation is UPDATE. IBM DB2 DBMSs always return one of the following values: • SQL_RESTRICT • SQL_NO_ACTION Both of these values indicate that an update is rejected if it removes a primary key row that a foreign key references, or adds a value in a foreign key that is not present in the primary key. You might encounter the following UPDATE_RULE values when connected to non-IBM RDBMSs: • SQL_CASCADE • SQL_SET_NULL
11	DELETE_RULE	SMALLINT	Identifies the action that is applied to the foreign key when the SQL operation is DELETE. The following values indicate the action that is applied: • SQL_CASCADE: when a primary key value is deleted, that value in related foreign keys is also deleted. • SQL_NO_ACTION: the delete is rejected if it removes values from a primary key that a foreign key references. • SQL_RESTRICT: the delete is rejected if it removes values from a primary key that a foreign key references. • SQL_SET_DEFAULT: when a primary key value is deleted, that value is replaced with a default value in related foreign keys. • SQL_SET_NULL: when a primary key value is deleted, that value is replaced with a null value in related foreign keys.
12	FK_NAME	VARCHAR(128)	Contains the name of the foreign key. This column contains a null value if it is not applicable to the data source.
13	PK_NAME	VARCHAR(128)	Contains the name of the primary key. This column contains a null value if it is not applicable to the data source.
14	DEFERRABILITY	SMALLINT	DB2 ODBC always returns a value of NULL. Other DBMSs support the following values: • SQL_INITIALLY_DEFERRED • SQL_INITIALLY_IMMEDIATE • SQL_NOT_DEFERRABLE

If you request foreign keys that are associated with a primary key, the returned rows in the result set are sorted by the values that the following columns contain:

1. FKTABLE_CAT
2. FKTABLE_SCHEM

3. FKTABLE_NAME
4. KEY_SEQ

If you request the primary keys that are associated with a foreign key, the returned rows in the result set are sorted by the values that the following columns contain:

1. PKTABLE_CAT
2. PKTABLE_SCHEM
3. PKTABLE_NAME
4. KEY_SEQ

The column names used by DB2 ODBC 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.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will remain unchanged.

DB2 ODBC applications that issue SQLForeignKeys() against a DB2 UDB for z/OS server should expect the result set columns listed in Table 85 on page 179.

For consistency with SQL92 limits, the VARCHAR columns of the result set are declared with a maximum length attribute of 128 bytes. Because DB2 names are smaller than 128 characters, you can always use a 128-character (plus the nul-terminator) output buffer to handle table names. Call SQLGetInfo() with each of the following attributes to determine the actual amount of space that you need to allocate when you connect to another DBMS:

- SQL_MAX_CATALOG_NAME_LEN to determine the length that the PKTABLE_CAT and FKTABLE_CAT columns support
- SQL_MAX_SCHEMA_NAME_LEN to determine the length that the PKTABLE_SCHEM and FKTABLE_SCHEM columns support
- SQL_MAX_TABLE_NAME_LEN to determine the length that the PKTABLE_NAME and FKTABLE_NAME columns support
- SQL_MAX_COLUMN_NAME_LEN to determine the length that the PKCOLUMN_NAME and FKCOLUMN_NAME columns support

Return codes

After you call SQLForeignKeys(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 86 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 86. SQLForeignKeys() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is open on the statement handle.

Table 86. SQLForeignKeys() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	The arguments *szPkTableName* and *szFkTableName* are both null pointers.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY090	Invalid string or buffer length.	This SQLSTATE is returned for one or more of the following reasons: • The value of one of the name length arguments is less than 0 and not equal SQL_NTS. • The length of the table or owner name is greater than the maximum length that is supported by the server. See "SQLGetInfo() - Get general information" on page 234.
HYC00	Driver not capable.	DB2 ODBC does not support ″catalog″ as a qualifier for table name.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.

Restrictions

None.

Example

Figure 17 on page 183 shows an application that uses SQLForeignKeys() to retrieve foreign key information about a table.

```
/******************************************************************/
 /*  Invoke SQLForeignKeys against PARENT Table. Find all          */
 /*  tables that contain foreign keys on PARENT.                   */
 /******************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "cli.h"
#include "sqlcli1.h"
#include "sqlcli1.h"

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            pTable [200];
   char            *pDSN = "STLEC1";
   SQLSMALLINT     update_rule;
   SQLSMALLINT     delete_rule;
   SQLINTEGER      update_rule_ind;
   SQLINTEGER      delete_rule_ind;
   char            update [25];
   char            delet  [25];
   typedef struct varchar    // define VARCHAR type
   {
     SQLSMALLINT  length;
     SQLCHAR      name [128];
     SQLINTEGER   ind;

   } VARCHAR;
   VARCHAR pktable_schem;
   VARCHAR pktable_name;
   VARCHAR pkcolumn_name;
   VARCHAR fktable_schem;
   VARCHAR fktable_name;
   VARCHAR fkcolumn_name;

   (void) printf ("**** Entering CLIP02.\n\n");

  /*****************************************************************/
  /* Allocate environment handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate connection handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocHandle( SQL_HANDLE_DBC, hEnv, &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a connect handle
     goto dberror;
```

Figure 17. An application that retrieves foreign key information about a table (Part 1 of 5)

```
/*****************************************************************/
  /* CONNECT TO data source (STLEC1)                              */
  /****************************************************************/

   RETCODE = SQLConnect(hDbc,        // Connect handle
                     (SQLCHAR *) pDSN, // DSN
                     SQL_NTS,     // DSN is nul-terminated
                     NULL,        // Null UID
                     0   ,
                     NULL,        // Null Auth string
                     0);

   if( RETCODE != SQL_SUCCESS )      // Connect failed
     goto dberror;

  /****************************************************************/
  /* Allocate statement handle                                    */
  /****************************************************************/

  rc = SQLAllocHandle( SQL_HANDLE_STMT, hDbc, &hStmt);

  if (rc != SQL_SUCCESS)
    goto exit;

  /****************************************************************/
  /* Invoke SQLForeignKeys against PARENT Table, specifying NULL   */
  /* for table with foreign key.                                   */
  /****************************************************************/

  rc = SQLForeignKeys (hStmt,
                      NULL,
                      0,
                      (SQLCHAR *) "ADMF001",
                      SQL_NTS,
                      (SQLCHAR *) "PARENT",
                      SQL_NTS,
                      NULL,
                      0,
                      NULL,
                      SQL_NTS,
                      NULL,
                      SQL_NTS);

  if (rc != SQL_SUCCESS)
  {
    (void) printf ("**** SQLForeignKeys Failed.\n");
    goto dberror;
  }
```

Figure 17. An application that retrieves foreign key information about a table (Part 2 of 5)

```
/******************************************************************/
/* Bind following columns of answer set:                          */
/*                                                                */
/*   2) pktable_schem                                             */
/*   3) pktable_name                                              */
/*   4) pkcolumn_name                                             */
/*   6) fktable_schem                                             */
/*   7) fktable_name                                              */
/*   8) fkcolumn_name                                             */
/*  10) update_rule                                               */
/*  11) delete_rule                                               */
/*                                                                */
/******************************************************************/

rc = SQLBindCol (hStmt,          // bind pktable_schem
                 2,
                 SQL_C_CHAR,
                 (SQLPOINTER) pktable_schem.name,
                 128,
                 &pktable_schem.ind);

rc = SQLBindCol (hStmt,          // bind pktable_name
                 3,
                 SQL_C_CHAR,
                 (SQLPOINTER) pktable_name.name,
                 128,
                 &pktable_name.ind);

rc = SQLBindCol (hStmt,          // bind pkcolumn_name
                 4,
                 SQL_C_CHAR,
                 (SQLPOINTER) pkcolumn_name.name,
                 128,
                 &pkcolumn_name.ind);
rc = SQLBindCol (hStmt,          // bind fktable_schem
                 6,
                 SQL_C_CHAR,
                 (SQLPOINTER) fktable_schem.name,
                 128,
                 &fktable_schem.ind);

rc = SQLBindCol (hStmt,          // bind fktable_name
                 7,
                 SQL_C_CHAR,
                 (SQLPOINTER) fktable_name.name,
                 128,
                 &fktable_name.ind);

rc = SQLBindCol (hStmt,          // bind fkcolumn_name
                 8,
                 SQL_C_CHAR,
                 (SQLPOINTER) fkcolumn_name.name,
                 128,
                 &fkcolumn_name.ind);

rc = SQLBindCol (hStmt,          // bind update_rule
                 10,
                 SQL_C_SHORT,
                 (SQLPOINTER) &update_rule;
                 0,
                 &update_rule_ind);
```

Figure 17. An application that retrieves foreign key information about a table (Part 3 of 5)

```
rc = SQLBindCol (hStmt,          // bind delete_rule
                 11,
                 SQL_C_SHORT,
                 (SQLPOINTER) &delete_rule,
                 0,
                 &delete_rule_ind);


/*****************************************************************/
/* Retrieve all tables with foreign keys defined on PARENT       */
/*****************************************************************/

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Primary Table Schema is %s. Primary Table Name is %s.\n",
                 pktable_schem.name, pktable_name.name);
  (void) printf ("**** Primary Table Key Column is %s.\n",
                 pkcolumn_name.name);
  (void) printf ("**** Foreign Table Schema is %s. Foreign Table Name is %s.\n",
                 fktable_schem.name, fktable_name.name);
  (void) printf ("**** Foreign Table Key Column is %s.\n",
                 fkcolumn_name.name);

  if (update_rule == SQL_RESTRICT)     // isolate update rule
    strcpy (update, "RESTRICT");
  else
  if (update_rule == SQL_CASCADE)
    strcpy (update, "CASCADE");
  else
    strcpy (update, "SET NULL");
  if (delete_rule == SQL_RESTRICT)     // isolate delete rule
    strcpy (delet, "RESTRICT");
  else
  if (delete_rule == SQL_CASCADE)
    strcpy (delet, "CASCADE");
  else
  if (delete_rule == SQL_NO_ACTION)
    strcpy (delet, "NO ACTION");
  else
    strcpy (delet, "SET NULL");

  (void) printf ("**** Update Rule is %s. Delete Rule is %s.\n",
                 update, delet);
}

/*****************************************************************/
/* Deallocate statement handle                                   */
/*****************************************************************/

rc = SQLFreeHandle (SQL_HANDLE_STMT, hStmt);

/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/

 RETCODE = SQLDisconnect(hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
```

Figure 17. An application that retrieves foreign key information about a table (Part 4 of 5)

```
/*****************************************************************/
  /* Deallocate connection handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeHandle (SQL_HANDLE_DBC, hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Free environment handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeHandle (SQL_HANDLE_ENV, hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP02.\n\n");

   return RETCODE;
}
```

Figure 17. An application that retrieves foreign key information about a table (Part 5 of 5)

Related functions

The following functions relate to SQLForeignKeys() calls. Refer to the descriptions of these functions for more information about how you can use SQLForeignKeys() in your applications.

- "SQLPrimaryKeys() - Get primary key columns of a table" on page 314
- "SQLStatistics() - Get index and statistics information for a base table" on page 381

SQLFreeConnect() - Free a connection handle

Purpose

Table 87. SQLFreeConnect() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	Yes

In the current version of DB2 ODBC, SQLFreeHandle() replaces SQLFreeConnect(). See "SQLFreeHandle() - Free a handle" on page 190 for more information.

Although DB2 ODBC supports SQLFreeConnect() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLFreeConnect() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLFreeConnect   (SQLHDBC          hdbc);
```

Function arguments

Table 88 lists the data type, use, and description for each argument in this function.

Table 88. SQLFreeConnect() arguments

Data type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle

SQLFreeEnv() - Free an environment handle

Purpose

Table 89. SQLFreeEnv() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	Yes

In the current version of DB2 ODBC, SQLFreeHandle() replaces SQLFreeEnv(). See “SQLFreeHandle() - Free a handle” on page 190 for more information.

Although DB2 ODBC supports SQLFreeEnv() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLFreeEnv() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLFreeEnv       (SQLHENV          henv);
```

Function arguments

Table 90 lists the data type, use, and description for each argument in this function.

Table 90. SQLFreeEnv arguments

Data type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle

SQLFreeHandle() - Free a handle

Purpose

Table 91. SQLFreeHandle() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLFreeHandle() frees an environment handle, a connection handle, or a statement handle.

Syntax

```
SQLRETURN  SQLFreeHandle     (SQLSMALLINT      HandleType,
                              SQLHANDLE        Handle);
```

Function arguments

Table 92 lists the data type, use, and description for each argument in this function.

Table 92. SQLFreeHandle() arguments

Data type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	Specifies the type of handle to be freed by SQLFreeHandle(). You must specify one of the following values: • SQL_HANDLE_ENV to free the environment handle • SQL_HANDLE_DBC to free a connection handle • SQL_HANDLE_STMT to free a statement handle
SQLHANDLE	*Handle*	input	Specifies the name of the handle to be freed.

Usage

Use SQLFreeHandle() to free handles for environments, connections, and statements. After you free a handle, you no longer use that handle in your application.

- **Freeing an environment handle**

  You must free all connection handles before you free the environment handle. If you attempt to free the environment handle while connection handles remain, SQLFreeHandle() returns SQL_ERROR and the environment and any active connection remains valid.
- **Freeing a connection handle**

  You must both free all statement handles and call SQLDisconnect() on a connection before you free the handle for that connection. If you attempt to free a connection handle while statement handles remain for that connection, SQLFreeHandle() returns SQL_ERROR and the connection remains valid.
- **Freeing a statement handle**

  When you call SQLFreeHandle() to free a statement handle, all resources that a call to SQLAllocHandle() with a *HandleType* of SQL_HANDLE_STMT allocates are freed. When you call SQLFreeHandle() to free a statement with pending results, those results are deleted.

SQLDisconnect() automatically drops any statements open on the connection.

Return codes

After you call SQLFreeHandle(), it returns one of the following values:
- SQL_SUCCESS
- SQL_INVALID_HANDLE
- SQL_ERROR

If the *HandleType* is not a valid type, SQLFreeHandle() returns SQL_INVALID_HANDLE. If SQLFreeHandle() returns SQL_ERROR, the handle is still valid.

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 93 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 93. SQLFreeHandle() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (SQLFreeHandle() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
08003	Connection is closed.	The *HandleType* argument specifies SQL_HANDLE_DBC, but the communication link between DB2 ODBC and the data source failed before the function completed processing.
HY000	General error.	An error occurred for which no specific SQLSTATE exists. The error message that is returned by SQLGetDiagRec() in the buffer that the *MessageText* argument specifies, describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory that is required to support execution or completion of the function.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • If the *HandleType* argument is SQL_HANDLE_ENV, and at least one connection is in an allocated or connected state, you must call SQLDisconnect() and SQLFreeHandle() to disconnect and free each connection before you can free the environment handle. If the *HandleType* argument is SQL_HANDLE_DBC you must free all statement handles on the connection, and disconnect before you can free the connection handle. • If the *HandleType* argument specifies SQL_HANDLE_STMT, SQLExecute() or SQLExecDirect() is called with the statement handle, and return SQL_NEED_DATA. This function is called before data is sent for all data-at-execution parameters or columns. You must issue SQLCancel() to free the statement handle.
HY013	Unexpected memory handling error.	The *HandleType* argument is SQL_HANDLE_STMT and the function call can not be processed because the underlying memory objects can not be accessed. This error can result from low memory conditions.
HY506	Error closing a file.	An error is encountered when trying to close a temporary file.

Restrictions

None.

Example

Refer to the online sample program DSN8O3VP in the DSN810.SDSNSAMP data set or to the “DSN8O3VP sample application” on page 531.

Related functions

The following functions relate to SQLFreeHandle() calls. Refer to the descriptions of these functions for more information about how you can use SQLFreeHandle() in your applications.

- “SQLAllocHandle() - Allocate a handle” on page 72
- “SQLCancel() - Cancel statement” on page 97
- “SQLDisconnect() - Disconnect from a data source” on page 140
- “SQLGetDiagRec() - Get multiple field settings of diagnostic record” on page 221

SQLFreeStmt() - Free (or reset) a statement handle

Purpose

Table 94. SQLFreeStmt() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLFreeStmt() ends processing for a statement, to which a statement handle refers. You use this function to perform the following tasks:
- Close a cursor
- Drop the statement handle and free the DB2 ODBC resources that are associated with the statement handle.

Call SQLFreeStmt() after you execute an SQL statement and process the results.

Syntax

```
SQLRETURN   SQLFreeStmt      (SQLHSTMT          hstmt,
                              SQLUSMALLINT      fOption);
```

Function arguments

Table 95 lists the data type, use, and description for each argument in this function.

Table 95. SQLFreeStmt() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle that refers to the statement to be stopped.
SQLUSMALLINT	*fOption*	input	The following values specify the manner in which you free the statement handle: • SQL_UNBIND • SQL_RESET_PARAMS • SQL_CLOSE • SQL_DROP (Deprecated) See "Usage" for details about these values.

Usage

When you call SQLFreeStmt(), you set the *fOption* argument to one of the following options. SQLFreeStmt() performs different actions based upon which one of these options you specify.

SQL_UNBIND
All the columns that are bound by previous SQLBindCol() calls on this statement handle are released (the association between application variables or file references and result set columns is broken).

SQL_RESET_PARAMS
All the parameters that are set by previous SQLBindParameter() calls on this statement handle are released. (The association between application variables, or file references, and parameter markers in the SQL statement for the statement handle is broken.)

SQL_CLOSE
The cursor (if any) that is associated with the statement handle is closed and all pending results are discarded. You can reopen the cursor by calling SQLExecute() or SQLExecDirect() with the same or different values in the application variables (if any) that are bound to the statement handle. The cursor name is retained until the statement handle is dropped or the next successful SQLSetCursorName() call. If a cursor is not associated with the statement handle, this option has no effect. (In the case where no cursors exist, a warning or an error is **not** generated.)

You can also call the ODBC 3.0 API SQLCloseCursor() to close the cursor. See “SQLCloseCursor() - Close a cursor and discard pending results” on page 99 for more information.

SQL_DROP (Deprecated)
In ODBC 3.0, SQLFreeHandle() with *HandleType* set to SQL_HANDLE_STMT replaces the SQL_DROP option of SQLFreeStmt(). See “SQLFreeHandle() - Free a handle” on page 190 for more information.

Although DB2 ODBC supports the SQL_DROP option for backward compatibility, you should use current ODBC 3.0 functions in your applications.

SQLFreeStmt() does not affect LOB locators. To free a locator, call SQLExecDirect() with the FREE LOCATOR statement. See “Using large objects” on page 423 for more information about LOBs.

After you execute a statement on a statement handle, you can reuse that handle to execute a different statement. The following situations require you to take additional action before you reuse a statement handle:

- When the statement handle that you want to reuse is associated with a catalog function or SQLGetTypeInfo(), you must close the cursor on that handle.
- When you want to reuse a statement handle for a different number or different types of parameters than you originally bound, you must reset the parameters on that handle.
- When you want to reuse a statement handle for a different number or different types of columns than you originally bound, you must unbind the original columns.

Alternatively, you can drop the statement handle and allocate a new one.

Return codes

After you call SQLFreeStmt(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

SQL_SUCCESS_WITH_INFO is not returned if *fOption* is set to SQL_DROP, because no statement handle is available to use when SQLGetDiagRec() is called.

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 96 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 96. SQLFreeStmt() SQLSTATEs

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY092	Option type out of range.	The specified value for the *fOption* argument is not one of the following values: • SQL_CLOSE • SQL_DROP • SQL_UNBIND • SQL_RESET_PARAMS

Restrictions

None.

Example

See Figure 16 on page 176.

Related functions

The following functions relate to SQLFreeStmt() calls. Refer to the descriptions of these functions for more information about how you can use SQLFreeStmt() in your applications.

- “SQLAllocHandle() - Allocate a handle” on page 72
- “SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator” on page 85
- “SQLExtendedFetch() - Fetch an array of rows” on page 163
- “SQLFetch() - Fetch the next row” on page 171
- “SQLSetParam() - Bind a parameter marker to a buffer” on page 364

SQLGetConnectAttr() - Get current attribute setting

Purpose

Table 97. SQLGetConnectAttr() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLGetConnectAttr() returns the current setting of a connection attribute. To set these attributes use SQLSetConnectAttr().

Syntax

```
SQLRETURN  SQLGetConnectAttr (SQLHDBC          ConnectionHandle,
                              SQLINTEGER       Attribute,
                              SQLPOINTER       ValuePtr,
                              SQLINTEGER       BufferLength,
                              SQLINTEGER       *StringLengthPtr);
```

Function arguments

Table 98 lists the data type, use, and description for each argument in this function.

Table 98. SQLGetConnectAttr() arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Specifies the connection handle from which you retrieve the attribute value.
SQLINTEGER	*Attribute*	input	Specifies the connection attribute to retrieve. Refer to Table 187 on page 347 for a complete list of attributes.
SQLPOINTER	*ValuePtr*	input	Specifies the pointer to the memory in which to return the current value of the attribute that the *Attribute* argument indicates. **ValuePtr* will be a 32-bit unsigned integer value or point to a nul-terminated character string. If the *Attribute* argument is a driver-specific value, the value in **ValuePtr* might be a signed integer.
SQLINTEGER	*BufferLength*	input	Specifies the size, in bytes, of the buffer to which the **ValuePtr* argument points. This argument behaves differently according to the following types of attributes: • For ODBC-defined attributes: – If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*. – If *ValuePtr* points to an integer, *BufferLength* is ignored. • For driver-defined attributes (IBM extension): – If *ValuePtr* points to a character string, this argument should be the length, in bytes, of **ValuePtr* or SQL_NTS. If SQL_NTS, the driver assumes that the length of **ValuePtr* is SQL_MAX_OPTIONS_STRING_LENGTH bytes (excluding the nul-terminator). – If *ValuePtr* points to an integer, *BufferLength* is ignored.

Table 98. SQLGetConnectAttr() arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER *	*StringLengthPtr*	output	Specifies a pointer to the buffer in which to return the total number of bytes (excluding the nul-termination character) that the *ValuePtr* argument requires. The following conditions apply to the *StringLengthPtr* argument: • 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 or equal to the value that is specified for the *BufferLength* argument, the data in *ValuePtr* is truncated to that specified value minus the length of a nul-termination character. DB2 ODBC nul-terminates the truncated data. • If the *Attribute* argument does not denote a string, DB2 ODBC ignores the *BufferLength* argument, and it does not return a value into the buffer to which the *StringLengthPtr* argument points.

Usage

Use SQLGetConnectAttr() to retrieve the value of a connection attribute that is set on a connection handle.

Although you can use SQLSetConnectAttr() to set attribute values for a statement handle, you cannot use SQLGetConnectAttr() to retrieve current attribute values for a statement handle. To retrieve statement attribute values, call SQLGetStmtAttr(). For a list of valid connection attributes, refer to Table 187 on page 347.

Return codes

After you call SQLGetConnectAttr(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 99 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 99. SQLGetConnectAttr() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	An informational message. (SQLGetConnectAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
01004	Data truncated.	The data that is returned in the buffer that the *ValuePtr* argument specifies is truncated. The length to which the data is truncated is equal to the value that is specified in the *BufferLength* argument, minus the length of a nul-termination character. The *StringLengthPtr* argument specifies a buffer that receives the size of the non-truncated string. (SQLGetConnectAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)

Table 99. SQLGetConnectAttr() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
08003	Connection is closed.	The *Attribute* argument specifies a value that requires an open connection, but the connection handle was not in a connected state.
HY000	General error.	An error occurred for which no specific SQLSTATE exists. The error message that SQLGetDiagRec() returns in the buffer that the *MessageText* argument specifies, describes this error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory that is required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the *BufferLength* argument is less than 0.
HY092	Option type out of range.	The specified value for the *Attribute* argument is not valid for this version of DB2 ODBC.
HYC00	Driver not capable.	The specified value for the *Attribute* argument is a valid connection or statement attribute for this version of the DB2 ODBC driver, but it is not supported by the data source.

Restrictions

None.

Example

The following example prints the current setting of a connection attribute. SQLGetConnectAttr() retrieves the current value of the SQL_ATTR_AUTOCOMMIT statement attribute.

```
SQLINTEGER output_nts,autocommit;
rc = SQLGetConnectAttr( hdbc, SQL_AUTOCOMMIT,
                        &autocommit, 0, NULL ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;
printf( "\nAutocommit is: " ) ;
if ( autocommit == SQL_AUTOCOMMIT_ON )
 printf( "ON\n" ) ;
else
 printf( "OFF\n" ) ;
```

Related functions

The following functions relate to SQLGetConnectAttr() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetConnectAttr() in your applications.

- “SQLGetStmtAttr() - Get current setting of a statement attribute” on page 272
- “SQLSetConnectAttr() - Set connection attributes” on page 346
- “SQLSetStmtAttr() - Set statement attributes” on page 367

SQLGetConnectOption() - Return current setting of a connect option

Purpose

Table 100. SQLGetConnectOption() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	No

In the current version of DB2 ODBC, SQLGetConnectAttr() replaces SQLGetConnectOption(). See “SQLAllocHandle() - Allocate a handle” on page 72 for more information.

Although DB2 ODBC supports SQLGetConnectOption() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLGetConnectOption() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLGetConnectOption (
                              SQLHDBC          hdbc,
                              SQLUSMALLINT     fOption,
                              SQLPOINTER       pvParam);
```

Function arguments

Table 101 lists the data type, use, and description for each argument in this function.

Table 101. SQLGetConnectOption() arguments

Data type	Argument	Use	Description
HDBC	*hdbc*	input	Connection handle.
SQLUSMALLINT	*fOption*	input	Attribute to set. See Table 187 on page 347 for the complete list of connection attributes and their descriptions.
SQLPOINTER	*pvParam*	input, output, or input and output	Value that is associated with the *fOption* argument. Depending on the value of the *fOption* argument, this can be a 32-bit integer value, or a pointer to a nul-terminated character string. The maximum length of any character string returned is SQL_MAX_OPTION_STRING_LENGTH bytes (which excludes the nul-terminator).

SQLGetCursorName() - Get cursor name

Purpose

Table 102. SQLGetCursorName() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLGetCursorName() returns the name of the cursor that is associated with a statement handle. If you explicitly set a cursor name with SQLSetCursorName(), the name that you specified in a call to SQLSetCursorName() is returned. If you do not explicitly set a name, SQLGetCursorName() returns the implicitly generated name for that cursor.

Syntax

```
SQLRETURN   SQLGetCursorName (SQLHSTMT         hstmt,
                              SQLCHAR     FAR   *szCursor,
                              SQLSMALLINT       cbCursorMax,
                              SQLSMALLINT FAR   *pcbCursor);
```

Function arguments

Table 103 lists the data type, use, and description for each argument in this function.

Table 103. SQLGetCursorName() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle on which the cursor you want to identify is open.
SQLCHAR *	*szCursor*	output	Specifies the buffer in which the cursor name is returned.
SQLSMALLINT	*cbCursorMax*	input	Specifies the size of the buffer to which the *szCursor* argument points.
SQLSMALLINT *	*pcbCursor*	output	Points to the buffer that receives the number of bytes that the cursor name requires.

Usage

SQLGetCursorName() returns the name that you set explicitly on a cursor with SQLSetCursorName(). If you do not set a name for a cursor, you can use this function to retrieve the name that DB2 ODBC internally generates.

SQLGetCursorName() returns the same cursor name (which can be explicit or implicit) on a statement until you drop that statement, or until you set another explicit name for that cursor. Cursor names are always 18 characters or less, and are always unique within a connection.

Cursor names that DB2 ODBC generates internally always begin with SQLCUR or SQL_CUR. For query result sets, DB2 ODBC also reserves SQLCURQRS as a cursor name prefix. (See “Restrictions” on page 201 for more details about this naming convention.)

Return codes

After you call SQLGetCursorName(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 104 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 104. SQLGetCursorName() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The cursor name that is returned in the buffer that the *szCursor* argument specifies is longer than the value in the *cbCursorMax* argument. Data in this buffer is truncated to the one byte less than the value that the *cbCursorMax* argument specifies. The *pcbCursor* argument contains the length, in bytes, that the full cursor name requires. (SQLGetCursorName() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY015	No cursor name available.	No cursor is open on the statement handle that the *hstmt* argument specifies, and no cursor name is set with SQLSetCursorName().
HY090	Invalid string or buffer length.	The value specified for the *cbCursorMax*argument is less than 0.
HY092	Option type out of range.	The statement handle specified for the *hstmt* argument is not valid.

Restrictions

ODBC generates cursor names that begin with SQL_CUR. X/Open CLI generates cursor names that begin with either SQLCUR or SQL_CUR.

DB2 ODBC is inconsistent with the ODBC specification for naming cursors. DB2 ODBC generates cursor names that begin with SQLCUR or SQL_CUR, which is consistent with the X/Open CLI standard.

Example

Figure 18 on page 202 shows an application that uses SQLGetCursorName() to extract the name of a cursor needed that the proceeding update statement requires.

```
 /******************************************************************/
 /*  Perform a positioned update on a column of a cursor.          */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT, hStmt2 = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS, rc2 = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            *pDSN = "STLEC1";

   SWORD           cbCursor;
   SDWORD          cbValue1;
   SDWORD          cbValue2;
   char            employee [30];
   int             salary = 0;
   char            cursor_name [20];
   char            update [200];

   char            *stmt = "SELECT NAME, SALARY FROM EMPLOYEE WHERE
                            SALARY > 100000 FOR UPDATE OF SALARY";
```

Figure 18. An application that extracts a cursor name (Part 1 of 5)

```
 (void) printf ("**** Entering CLIP04.\n\n");

/*****************************************************************/
/* Allocate environment handle                                   */
/*****************************************************************/

 RETCODE = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;

/*****************************************************************/
/* Allocate connection handle to DSN                             */
/*****************************************************************/

 RETCODE = SQLAllocHandle( SQL_HANDLE_DBC, hEnv, &hDbc);

 if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
   goto dberror;

/*****************************************************************/
/* CONNECT TO data source (STLEC1)                               */
/*****************************************************************/

 RETCODE = SQLConnect(hDbc,        // Connect handle
                     (SQLCHAR *) pDSN, // DSN
                     SQL_NTS,     // DSN is nul-terminated
                     NULL,        // Null UID
                     0   ,
                     NULL,        // Null Auth string
                     0);

 if( RETCODE != SQL_SUCCESS )      // Connect failed
   goto dberror;

/*****************************************************************/
/* Allocate statement handles                                    */
/*****************************************************************/

rc = SQLAllocHandle( SQL_HANDLE_STMT, hDbc, &hStmt);

if (rc != SQL_SUCCESS)
  goto exit;
```

Figure 18. An application that extracts a cursor name (Part 2 of 5)

```
rc = SQLAllocHandle( SQL_HANDLE_STMT, hDbc, &hStmt2);

if (rc != SQL_SUCCESS)
  goto exit;

/*****************************************************************/
/* Execute query to retrieve employee names                      */
/*****************************************************************/

rc = SQLExecDirect (hStmt,
                    (SQLCHAR *) stmt,
                    strlen(stmt));

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** EMPLOYEE QUERY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Extract cursor name -- required to build UPDATE statement.    */
/*****************************************************************/

rc = SQLGetCursorName (hStmt,
                       (SQLCHAR *) cursor_name,
                       sizeof(cursor_name),
                       &cbCursor);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** GET CURSOR NAME FAILED.\n");
  goto dberror;
}

(void) printf ("**** Cursor name is %s.\n");

rc = SQLBindCol (hStmt,           // bind employee name
                 1,
                 SQL_C_CHAR,
                 employee,
                 sizeof(employee),
                 &cbValue1);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF NAME FAILED.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind employee salary
                 2,
                 SQL_C_LONG,
                 &salary,
                 0,
                 &cbValue2);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF SALARY FAILED.\n");
  goto dberror;
}
```

Figure 18. An application that extracts a cursor name (Part 3 of 5)

```
/******************************************************************/
/* Answer set is available -- Fetch rows and update salary       */
/******************************************************************/

while (((rc = SQLFetch (hStmt)) == SQL_SUCCESS) &&;
       (rc2 == SQL_SUCCESS))
{
  int new_salary = salary*1.1;

  (void) printf ("**** Employee Name %s with salary %d. New salary = %d.\n",
                 employee,
                 salary,
                 new_salary);

  sprintf (update,
          "UPDATE EMPLOYEE SET SALARY = %d WHERE CURRENT OF %s",
          new_salary,
          cursor_name);

  (void) printf ("***** Update statement is : %s\n", update);

  rc2 = SQLExecDirect (hStmt2,
                       (SQLCHAR *) update,
                       SQL_NTS);
}

if (rc2 != SQL_SUCCESS)
{
  (void) printf ("**** EMPLOYEE UPDATE FAILED.\n");
  goto dberror;
}

/******************************************************************/
/* Reexecute query to validate that salary was updated           */
/******************************************************************/

rc = SQLCloseCursor(hStmt);

rc = SQLExecDirect (hStmt,
                    (SQLCHAR *) stmt,
                    strlen(stmt));

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** EMPLOYEE QUERY FAILED.\n");
  goto dberror;
}

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Employee Name %s has salary %d.\n",
                 employee,
                 salary);
}
```

Figure 18. An application that extracts a cursor name (Part 4 of 5)

```
  /*****************************************************************/
  /* Deallocate statement handles                                  */
  /*****************************************************************/

  rc = SQLFreeHandle (SQL_HANDLE_STMT, hStmt);

  rc = SQLFreeHandle (SQL_HANDLE_STMT, hStmt2);

  /*****************************************************************/
  /* DISCONNECT from data source                                   */
  /*****************************************************************/

   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Deallocate connection handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeHandle (SQL_HANDLE_DBC, hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Free environment handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeHandle (SQL_HANDLE_ENV, hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP04.\n\n");

   return RETCODE;
}
```

Figure 18. An application that extracts a cursor name (Part 5 of 5)

Related functions

The following functions relate to SQLGetCursorName() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetCursorName() in your applications.

- “SQLExecute() - Execute a statement” on page 160
- “SQLExecDirect() - Execute a statement directly” on page 154
- “SQLPrepare() - Prepare a statement” on page 306
- “SQLSetCursorName() - Set cursor name” on page 357

SQLGetData() - Get data from a column

Purpose

Table 105. SQLGetData() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLGetData() retrieves data for a single column in the current row of the result set. Using this function is an alternative to using SQLBindCol(), which transfers data directly into application variables or LOB locators on each SQLFetch() or SQLExtendedFetch() call. You can also use SQLGetData() to retrieve large data values in pieces.

You must call SQLFetch() before SQLGetData().

After you call SQLGetData() for each column, call SQLFetch() or SQLExtendedFetch() for each row you want to retrieve.

Syntax

```
SQLRETURN   SQLGetData       (SQLHSTMT          hstmt,
                              SQLUSMALLINT      icol,
                              SQLSMALLINT       fCType,
                              SQLPOINTER        rgbValue,
                              SQLINTEGER        cbValueMax,
                              SQLINTEGER  FAR   *pcbValue);
```

Function arguments

Table 106 lists the data type, use, and description for each argument in this function.

Table 106. SQLGetData() arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle on which the result set is generated.
SQLUSMALLINT	*icol*	input	Specifies the column number of the result set for which the data retrieval is requested.

Table 106. SQLGetData() arguments (continued)

Data Type	Argument	Use	Description
SQLSMALLINT	*fCType*	input	Specifies the C data type of the column that *icol* indicates. You can specify the following types for the *fCType* argument: • 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_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_WCHAR When you specify SQL_C_DEFAULT, data is converted to its default C data type; see Table 4 on page 25 for more information.
SQLPOINTER	*rgbValue*[1]	output	Points to a buffer where the retrieved column data is stored.
SQLINTEGER	*cbValueMax*	input	Specifies the maximum size of the buffer to which the *rgbValue* argument points.
SQLINTEGER *	*pcbValue*[1]	output	Points to the value that indicates the amount of space that the data you are retrieving requires. If the data is 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, this function fails because it has no way to report that the data is null. If SQLFetch() fetches a column that contains binary data, then the pointer that the *pcbValue* argument specifies must not be null. SQLGetData() fails in this case because it cannot inform the application about the length of the data that is returned to the buffer that the *rgbValue* argument specifies.

Note:

1. DB2 ODBC provides some performance enhancement if the buffer that the *rgbValue* argument specifies is placed consecutively in memory after the value to which the *pcbValue.* argument points.

Usage

You can use SQLGetData() in combination with SQLBindCol() on the same result set, if you use SQLFetch(). **Do not** use SQLExtendedFetch(). Use the following procedure to retrieve data with SQLGetData():

1. Call SQLFetch(), which advances cursor to first row, retrieves first row, and transfers data for bound columns.
2. Call SQLGetData(), which transfers data for the specified column.
3. Repeat step 2 for each column needed.
4. Call SQLFetch(), which advances the cursor to the next row, retrieves the next row, and transfers data for bound columns.

5. Repeat steps 2, 3 and 4 for each row that is in the result set, or until the result set is no longer needed.

You can also use SQLGetData() to retrieve long columns if the C data type (which you specify with the *fCType* argument) is SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR, or if *fCType* is SQL_C_DEFAULT and the column type denotes a binary or character string.

Handling encoding schemes: The CURRENTAPPENSCH keyword in the DB2 ODBC initialization file and the *fCType* argument in SQLGetData() determines which one of the following encoding schemes is used for character and graphic data.

- The ODBC driver places EBCDIC data into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = EBCDIC is specified in the initialization file, or the CURRENTAPPENSCH keyword is not specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLGetData() call.
- The ODBC driver places Unicode UCS-2 data into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = UNICODE is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_WCHAR in the SQLGetData() call.
- The ODBC driver places Unicode UTF-8 data into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = UNICODE is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR in the SQLGetData() call.
- The ODBC driver places ASCII data into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = ASCII is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLGetData() call.

For more information about encoding schemes, see "Handling application encoding schemes" on page 443.

Handling data truncation: After each SQLGetData() call, if the data available for return is greater than or equal to *cbValueMax*, the data is truncated. Truncation is indicated by a function return code of SQL_SUCCESS_WITH_INFO coupled with a SQLSTATE denoting data truncation. You can call SQLGetData() again, on the same column, to subsequently retrieve the truncated data. To obtain the entire column, repeat these calls until SQLGetData() returns SQL_SUCCESS. If you call SQLGetData() after it returns SQL_SUCCESS, it returns SQL_NO_DATA_FOUND.

When DB2 ODBC truncates digits to the right of the decimal point from numeric data types, DB2 ODBC issues a warning. When DB2 ODBC truncates digits to the left of the decimal point, however, DB2 ODBC returns an error. (See "Diagnostics" on page 211 for more information.)

To eliminate warnings when data is truncated, call SQLSetStmtAttr() with the SQL_ATTR_MAX_LENGTH attribute set to a maximum length value. Then allocate a buffer for the *rgbValue* argument that is the same size (plus the nul-terminator) as the value that you specified for SQL_ATTR_MAX_LENGTH. If the column data is larger than the maximum number of bytes that you specified for

SQL_ATTR_MAX_LENGTH, SQL_SUCCESS is returned. When you specify a maximum length, DB2 ODBC returns the length you specify, not the actual length, for the *pcbValue* argument.

Using LOB locators: Although you can use SQLGetData() to retrieve LOB column data sequentially, use the DB2 ODBC LOB functions when you need a only portion or a few sections of LOB data. Use the following procedure instead of SQLGetData() if you want to retrieve portions of LOB values:

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() might also be required for determining the values of some of the arguments).
4. Repeat step 2 and 3 for each row in the result set.

Discarding data from an active retrieval: To discard data from a retrieval that is currently active, call SQLGetData() with the *icol* argument set to the next column position from which you want to retrieve data. To discard data that you have not retrieved, call SQLFetch() to advance the cursor to the next row. Call SQLFreeStmt() or SQLCloseCursor() if you have finished retrieving data from the result set.

Allocating buffers: The *fCType* input argument determines the type of data conversion (if any) that occurs before the column data is placed into the buffer to which the *rgbValue* argument points.

For SQL graphic column data, the following conditions apply:

- The size of the buffer that the *rgbValue* argument specifies must be a multiple of 2 bytes. (The *cbValueMax* value must specify this value as a multiple of 2 bytes also.) Before you call SQLGetData(), call SQLDescribeCol() or SQLColAttribute to determine the SQL data type and the length, in bytes, of data in the result set.
- The *pcbValue* argument must not specify a null pointer. DB2 ODBC stores the number of octets that are stored in the buffer to which the *rgbValue* argument points.
- If you retrieve data in pieces, DB2 ODBC attempts to fill *rgbValue* to the nearest multiple of two octets that is less than or equal to the value the *cbValueMax* argument specifies. If *cbValueMax* is not a multiple of two, the last byte in that buffer is never used. DB2 ODBC does not split a double-byte character.

The buffer that the *rgbValue* argument specifies contains nul-terminated values, unless you retrieve binary data, or the SQL data type of the column is graphic (DBCS) and the C buffer type is SQL_C_CHAR. If you retrieve data in pieces, you must perform the proper adjustments to the nul-terminator when you reassemble these pieces. (That is, you must remove nul-terminators before concatenating the pieces of data.)

Return codes

After you call SQLGetData(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_SUCCESS is returned if SQLGetData() retrieves a zero-length string. For zero-length strings, *pcbValue* contains 0, and *rgbValue* contains a nul-terminator.

SQL_NO_DATA_FOUND is returned when the preceding SQLGetData() call has retrieved all of the data for this column.

For a description of each return code value, see "Function return codes" on page 23.

If the preceding call to SQLFetch() failed, **do not** call SQLGetData(). In this case, SQLGetData() retrieves undefined data.

Diagnostics

Table 107 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 107. SQLGetData() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	Data that is returned for the column that the *icol* argument specifies is truncated. String or numeric values are right truncated. (SQLGetData() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
07006	Invalid conversion.	This SQLSTATE is returned for one or more of the following reasons: • The data value cannot be converted to the C data type specified by the *fCType* argument. • The function is called with a value for the *icol* argument that was specified in a previous SQLGetData() call, but the value for the *fCType* argument differs in each of these calls.
22002	Invalid output or indicator buffer specified.	The pointer that is specified in the *pcbValue* argument is a null pointer, and the value of the column is also null. The function cannot report SQL_NULL_DATA.
22008	Invalid datetime format or datetime field overflow.	Datetime field overflow occurred. **Example:** An arithmetic operation on a date or timestamp results in a value that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because the variable is too small.
22018	Error in assignment.	A returned value is incompatible with the data type that the *fCType* argument denotes.
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 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.

Table 107. SQLGetData() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY002	Invalid column number.	This SQLSTATE is returned for one or more of the following reasons: • The specified column is less than 0 or greater than the number of result columns. • The specified column is 0 (the *icol* argument is set to 0), but DB2 ODBC does not support ODBC bookmarks. • SQLExtendedFetch() is called for this result set.
HY003	Program type out of range.	The *fCType*argument specifies an invalid data type or SQL_C_DEFAULT.
HY009	Invalid use of a null pointer.	This SQLSTATE is returned for one or more of the following reasons: • The *rgbValue* argument specifies a null pointer. • The *pcbValue* argument specifies a null pointer but the SQL data type of the column is graphic (DBCS). • The *pcbValue* argument specifies a null pointer but the *fCType* argument specifies SQL_C_CHAR.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • The statement handle does not contain a cursor in a positioned state. SQLGetData() is called without first calling SQLFetch(). • The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY019	Numeric value out of range.	When the numeric value (as numeric or string) for the column is returned, the whole part of the number is truncated.
HY090	Invalid string or buffer length.	The value of the *cbValueMax* argument is less than 0 and the*fCType* argument specifies one of the following values: • SQL_C_CHAR • SQL_C_BINARY • SQL_C_DBCHAR • SQL_C_DEFAULT (for the default types of SQL_C_CHAR, SQL_C_BINARY, or SQL_C_DBCHAR)
HYC00	Driver not capable.	This SQLSTATE is returned for one or more of the following reasons: • The SQL data type for the specified data type is recognized but DB2 ODBC does not support this data type. • DB2 ODBC cannot perform the conversion between the SQL data type and application data type that is specified in the *fCType* argument. • SQLExtendedFetch() is called on the statement handle that is specified in the *hstmt* argument.

Restrictions

ODBC has defined column 0 for bookmarks. DB2 ODBC does not support bookmarks.

Example

Figure 19 shows an application that uses SQLGetData() to retrieve data. See "Example" on page 175 for a comparison to use bound columns instead of SQLGetData().

```
 /******************************************************************/
 /*       Populate BIOGRAPHY table from flat file text. Insert    */
 /*       VITAE in 80-byte pieces via SQLPutData. Also retrieve   */
 /*       NAME, UNIT and VITAE for all members. VITAE is retrieved*/
 /*       via SQLGetData.                                         */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

#define TEXT_SIZE 80

int insert_bio (SQLHSTMT hStmt,       // insert_bio prototype
                char     *bio,
                int       bcount);

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT, hStmt2 = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   FILE            *fp;
   SQLINTEGER      RETCODE = 0;
   char            pTable [200];
   char            *pDSN = "STLEC1";
   UDWORD          pirow;
   SDWORD          cbValue;
   char            *i_stmt = "INSERT INTO BIOGRAPHY VALUES (?, ?, ?)";
   char            *query  = "SELECT NAME, UNIT, VITAE FROM BIOGRAPHY";
   char             text [TEXT_SIZE]; // file text
   char             vitae [3200];    // biography text
   char             Narrative [TEXT_SIZE];
   SQLINTEGER       vitae_ind = SQL_DATA_AT_EXEC; // bio data is
                                      // passed at execute time
                                      // via SQLPutData
   SQLINTEGER       vitae_cbValue = TEXT_SIZE;
   char             *t = NULL;
   char             *c = NULL;
   char             name [21];
   SQLINTEGER       name_ind = SQL_NTS;
   SQLINTEGER       name_cbValue = sizeof(name);
   char             unit [31];
   SQLINTEGER       unit_ind = SQL_NTS;
   SQLINTEGER       unit_cbValue = sizeof(unit);
   char             tmp [80];
   char             *token = NULL, *pbio = vitae;
   char             insert = SQL_FALSE;
   int              i, bcount = 0;

   (void) printf ("**** Entering CLIP09.\n\n");
```

Figure 19. An application that retrieves data with SQLGetData() (Part 1 of 8)

```
    /****************************************************************/
  /* Allocate environment handle                                  */
  /****************************************************************/

   RETCODE = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, hEnv, &hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /****************************************************************/
  /* Allocate connection handle to DSN                            */
  /****************************************************************/

   RETCODE = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;

  /****************************************************************/
  /* CONNECT TO data source (STLEC1)                              */
  /****************************************************************/

   RETCODE = SQLConnect(hDbc,        // Connect handle
                     (SQLCHAR *) pDSN, // DSN
                     SQL_NTS,     // DSN is nul-terminated
                     NULL,        // Null UID
                     0   ,
                     NULL,        // Null Auth string
                     0);

   if( RETCODE != SQL_SUCCESS )      // Connect failed
     goto dberror;

/****************************************************************/
  /* Allocate statement handles                                   */
  /****************************************************************/

   rc = SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt);

   if (rc != SQL_SUCCESS)
   {
     (void) printf ("**** Allocate statement handle failed.\n");
     goto dberror;
   }

  rc = SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt2);

   if (rc != SQL_SUCCESS)
   {
     (void) printf ("**** Allocate statement handle failed.\n");
     goto dberror;
   }
```

Figure 19. An application that retrieves data with SQLGetData() (Part 2 of 8)

```
/******************************************************************/
/* Prepare INSERT statement.                                      */
/******************************************************************/

 rc = SQLPrepare (hStmt,
                 (SQLCHAR *) i_stmt,
                 SQL_NTS);

 if (rc != SQL_SUCCESS)
 {
   (void) printf ("**** Prepare of INSERT failed.\n");
   goto dberror;
 }

/******************************************************************/
/* Bind NAME and UNIT. Bind VITAE so that data can be passed      */
/* via SQLPutData.                                                */
/******************************************************************/

 rc = SQLBindParameter (hStmt,        // bind NAME
                       1,
                       SQL_PARAM_INPUT,
                       SQL_C_CHAR,
                       SQL_CHAR,
                       sizeof(name),
                       0,
                       name,
                       sizeof(name),
                       &name_ind);

 if (rc != SQL_SUCCESS)
 {
   (void) printf ("**** Bind of NAME failed.\n");
   goto dberror;
 }
  rc = SQLBindParameter (hStmt,        // bind Branch
                       2,
                       SQL_PARAM_INPUT,
                       SQL_C_CHAR,
                       SQL_CHAR,
                       sizeof(unit),
                       0,
                       unit,
                       sizeof(unit),
                       &unit_ind);

 if (rc != SQL_SUCCESS)
 {
   (void) printf ("**** Bind of UNIT failed.\n");
   goto dberror;
 }

 rc = SQLBindParameter (hStmt,        // bind Rank
                       3,
                       SQL_PARAM_INPUT,
                       SQL_C_CHAR,
                       SQL_LONGVARCHAR,
                       3200,
                       0,
                       (SQLPOINTER) 3,
                       0,
                       &vitae_ind);
```

Figure 19. An application that retrieves data with SQLGetData() (Part 3 of 8)

```
     if (rc != SQL_SUCCESS)
     {
       (void) printf ("**** Bind of VITAE failed.\n");
       goto dberror;
     }

    /*****************************************************************/
    /* Read biographical text from flat file                         */
    /*****************************************************************/

     if ((fp = fopen ("DD:BIOGRAF", "r")) == NULL)  // open command file
     {
       rc = SQL_ERROR;                  // open failed
       goto exit;
     }

  /*****************************************************************/
  /* Process file and insert biographical text                     */
  /*****************************************************************/

     while (((t = fgets (text, sizeof(text), fp)) != NULL) &&;
            (rc == SQL_SUCCESS))
     {
       if (text[0] == #')      // if commander data
       {
         if (insert)                     // if BIO data to be inserted
         {
           rc = insert_bio (hStmt,
                            vitae,
                            bcount);     // insert row into BIOGRAPHY Table
           bcount = 0;                   // reset text line count
           pbio   = vitae;               // reset text pointer
         }
         token = strtok (text+1, ",");   // get member NAME
         (void) strcpy (name, token);
         token = strtok (NULL, "#");     // extract UNIT
         (void) strcpy (unit, token);    // copy to local variable
                                         // SQLPutData
         insert = SQL_TRUE;              // have row to insert
       }
       else
       {
         memset (pbio, ' ', sizeof(text));
         strcpy (pbio, text);            // populate text
         i = strlen (pbio);              // remove '\n' and '\0'
         pbio [i--] =' ';
         pbio [i]   =' ';
         pbio += sizeof (text);          // advance pbio
         bcount++;                       // one more text line
       }
     }

     if (insert)                         // if BIO data to be inserted
     {
       rc = insert_bio (hStmt,
                        vitae,
                        bcount);         // insert row into BIOGRAPHY Table
     }

     fclose (fp);                        // close text flat file
```

Figure 19. An application that retrieves data with SQLGetData() (Part 4 of 8)

```
/******************************************************************/
/* Commit insert of rows                                          */
/******************************************************************/

 rc =SQLEndTran(SQL_HANDLE_DBC, hDbc, SQL_COMMIT);
 if (rc != SQL_SUCCESS)
 {
   (void) printf ("**** COMMIT FAILED.\n");
   goto dberror;
 }

/******************************************************************/
/* Open query to retrieve NAME, UNIT and VITAE. Bind NAME and    */
/* UNIT but leave VITAE unbound. Retrieved using SQLGetData.     */
/******************************************************************/

 RETCODE = SQLPrepare (hStmt2,
                      (SQLCHAR *)query,
                      strlen(query));

 if (RETCODE != SQL_SUCCESS)
 {
   (void) printf ("**** Prepare of Query Failed.\n");
   goto dberror;
 }
  RETCODE = SQLExecute (hStmt2);

 if (RETCODE != SQL_SUCCESS)
 {
   (void) printf ("**** Query Failed.\n");
   goto dberror;
 }

 RETCODE = SQLBindCol (hStmt2,            // bind NAME
                      1,
                      SQL_C_DEFAULT,
                      name,
                      sizeof(name),
                      &name_cbValue);

 if (RETCODE != SQL_SUCCESS)
 {
   (void) printf ("**** Bind of NAME Failed.\n");
   goto dberror;
 }

 RETCODE = SQLBindCol (hStmt2,            // bind UNIT
                      2,
                      SQL_C_DEFAULT,
                      unit,
                      sizeof(unit),
                      &unit_cbValue);
```

Figure 19. An application that retrieves data with SQLGetData() (Part 5 of 8)

```
 if (RETCODE != SQL_SUCCESS)
 {
   (void) printf ("**** Bind of UNIT Failed.\n");
   goto dberror;
 }

 while ((RETCODE = SQLFetch (hStmt2)) != SQL_NO_DATA_FOUND)
 {
   (void) printf ("**** Name is %s. Unit is %s.\n\n", name, unit);
   (void) printf ("**** Vitae follows:\n\n");

   for (i = 0; (i < 3200 && RETCODE != SQL_NO_DATA_FOUND); i += TEXT_SIZE)
   {
     RETCODE = SQLGetData (hStmt2,
                           3,
                           SQL_C_CHAR,
                           Narrative,
                           sizeof(Narrative) + 1,
                           &vitae_cbValue);

     if (RETCODE != SQL_NO_DATA_FOUND)
       (void) printf ("%s\n", Narrative);
   }
 }

/*****************************************************************/
/* Deallocate statement handles                                  */
/*****************************************************************/

 rc = SQLAllocHandle(SQL_HANDLE_STMT, hDbc, hStmt);

 rc = SQLAllocHandle(SQL_HANDLE_STMT, hDbc, hStmt2);

/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/

 RETCODE = SQLDisconnect(hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;

/*****************************************************************/
/* Deallocate connection handle                                  */
/*****************************************************************/

 RETCODE = SQLFreeHandle(SQL_HANDLE_DBC, hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
```

Figure 19. An application that retrieves data with SQLGetData() (Part 6 of 8)

```
  /*****************************************************************/
  /* Free environment handle                                       */
  /*****************************************************************/

   rc = SQLFreeHandle (SQL_HANDLE_ENV, hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP09.\n\n");

   return RETCODE;
}

/*****************************************************************/
/* Function insert_bio is invoked to insert one row into the     */
/* BIOGRAPHY Table. The biography text is inserted in sets of    */
/* 80 bytes via SQLPutData.                                      */
/*****************************************************************/

int insert_bio (SQLHSTMT hStmt,
                char     *vitae,
                int       bcount)
{
  SQLINTEGER       rc = SQL_SUCCESS;
  SQLPOINTER       prgbValue;
  int              i;
  char             *text;

  /*****************************************************************/
  /* NAME and UNIT are bound... VITAE is provided after execution  */
  /* of the INSERT using SQLPutData.                               */
  /*****************************************************************/

  rc = SQLExecute (hStmt);

  if (rc != SQL_NEED_DATA)     // expect SQL_NEED_DATA
  {
    rc = 12;
    (void) printf ("**** NEED DATA not returned.\n");
    goto exit;
  }
  /*****************************************************************/
  /* Invoke SQLParamData to position ODBC driver on input parameter*/
  /*****************************************************************/

  if ((rc = SQLParamData (hStmt,
                          &prgbValue)) != SQL_NEED_DATA)
  {
    rc = 12;
    (void) printf ("**** NEED DATA not returned.\n");
    goto exit;
  }
```

Figure 19. An application that retrieves data with SQLGetData() (Part 7 of 8)

```
  /*****************************************************************/
  /* Iterate through VITAE in 80 byte increments.... pass to       */
  /* ODBC Driver via SQLPutData.                                   */
  /*****************************************************************/

  for (i = 0, text = vitae, rc = SQL_SUCCESS;
       (i < bcount) && (rc == SQL_SUCCESS);
       i++, text += TEXT_SIZE)
  {
    rc = SQLPutData (hStmt,
                     text,
                     TEXT_SIZE);
  }
  /*****************************************************************/
  /* Invoke SQLParamData to trigger ODBC driver to execute the     */
  /* statement.                                                    */
  /*****************************************************************/

  if ((rc = SQLParamData (hStmt,
                          &prgbValue)) != SQL_SUCCESS)
  {
    rc = 12;
    (void) printf ("**** INSERT Failed.\n");
  }
  exit:
  return (rc);
}
```

Figure 19. An application that retrieves data with SQLGetData() (Part 8 of 8)

Related functions

The following functions relate to SQLGetData() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetData() in your applications.

- “SQLExtendedFetch() - Fetch an array of rows” on page 163
- “SQLFetch() - Fetch the next row” on page 171

SQLGetDiagRec() - Get multiple field settings of diagnostic record

Purpose

Table 108. SQLGetDiagRec() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLGetDiagRec() returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information. 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 109 lists the data type, use, and description for each argument in this function.

Table 109. SQLGetDiagRec() arguments

Data type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	Specifies a handle type identifier that describes the type of handle that you diagnose. This argument must specify one of the following values: • SQL_HANDLE_ENV for environment handles • SQL_HANDLE_DBC for connection handles • SQL_HANDLE_STMT for statement handles
SQLHANDLE	*Handle*	input	Specifies a handle for the diagnostic data structure. This handle must be the type of handle that the *HandleType* argument indicates.
SQLSMALLINT	*RecNumber*	input	Indicates the status record from which the application seeks information. Status records are numbered from 1.
SQLCHAR *	*SQLState*	output	Points to a buffer in which the five-character SQLSTATE, which corresponds to the diagnostic record that is specified in the *RecNumber* argument, is returned. The first two characters of this SQLSTATE indicate the class; the next three characters indicate the subclass.
SQLINTEGER *	*NativeErrorPtr*	output	Points to a buffer in the native error code, which is specific to the data source, is returned.
SQLCHAR *	*MessageText*	output	Points to a buffer in which the error message text is returned. The fields returned by SQLGetDiagRec() are contained in a text string.
SQLSMALLINT	*BufferLength*	input	Length (in bytes) of the buffer that the *MessageText* argument specifies.

Table 109. SQLGetDiagRec() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT *	*TextLengthPtr*	output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes required for the nul-termination character) available to return in the buffer that the *MessageText* argument specifies. If the number of bytes available to return is greater than the value that the *BufferLength* argument specifies, the error message text in the buffer is truncated to the value specified for the *BufferLength* argument minus the length of a nul-termination character.

Usage

An application typically calls SQLGetDiagRec() when a previous call to a DB2 ODBC function has returned anything other than SQL_SUCCESS. However, because any function can post zero or more errors each time it is called, 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() retrieves only the diagnostic information most recently associated with the handle specified in the Handle argument. If the application calls any other function, except SQLGetDiagRec() (or the ODBC 2.0 SQLGetDiagRec() function), any diagnostic information from the previous calls on the same handle is lost.

An application can scan all diagnostic records by looping while it increments *RecNumber* as long as SQLGetDiagRec() returns SQL_SUCCESS.

Calls to SQLGetDiagRec() are nondestructive to the diagnostic 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 the ODBC 2.0 SQLGetDiagRec() function), has been called in the interim.

Return codes

After you call SQLGetDiagRec(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see the “Diagnostics.”

Diagnostics

SQLGetDiagRec() does not post error values. It uses the function return codes to report diagnostic information. When you call SQLGetDiagRec(), these return codes represent the diagnostic information:
- SQL_SUCCESS: The function successfully returned diagnostic information.
- SQL_SUCCESS_WITH_INFO: The buffer that to which the *MessageText* argument points is too small to hold the requested diagnostic message. No diagnostic records are generated. To determine whether truncation occurred, compare the value specified for the *BufferLength* argument to the actual number of bytes available, which is written to the buffer to which the *TextLengthPtr* argument points.

- SQL_INVALID_HANDLE: The handle indicated by *HandleType* and *Handle* is not a valid handle.
- SQL_ERROR: One of the following occurred:
  - The *RecNumber* argument is negative or 0.
  - The *BufferLength* argument is less than zero.
- SQL_NO_DATA: The *RecNumber* argument is greater than the number of diagnostic records that exist for the handle that is specified in the *Handle* argument. The function also returns SQL_NO_DATA for any positive value for the *RecNumber* argument if no diagnostic records are produced for the handle that the*Handle* argument specifies.

Restrictions

None.

Example

Refer to the sample program DSN8O3VP online in the data set DSN810.SDSNSAMP or to “DSN8O3VP sample application” on page 531.

Related functions

The following functions relate to SQLGetDiagRec() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetDiagRec() in your applications.
- “SQLFreeHandle() - Free a handle” on page 190
- “SQLFreeStmt() - Free (or reset) a statement handle” on page 193
- “SQLGetInfo() - Get general information” on page 234

SQLGetEnvAttr() - Return current setting of an environment attribute

Purpose

Table 110. SQLGetEnvAttr() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLGetEnvAttr() returns the current setting for an environment attribute. You use the SQLSetEnvAttr() function to set these attributes.

Syntax

```
SQLRETURN  SQLGetEnvAttr (SQLHENV          EnvironmentHandle,
                          SQLINTEGER       Attribute,
                          SQLPOINTER       ValuePtr,
                          SQLINTEGER       BufferLength,
                          SQLINTEGER       *StringLengthPtr);
```

Function arguments

Table 111 lists the data type, use, and description for each argument in this function.

Table 111. SQLGetEnvAttr() arguments

Data type	Argument	Use	Description
SQLHENV	*EnvironmentHandle*	input	Specifies the environment handle.
SQLINTEGER	*Attribute*	input	Specifies the attribute to retrieve. See Table 199 on page 361 for the list of environment attributes and their descriptions.
SQLPOINTER	*ValuePtr*	output	Points to the buffer in which the current value associated with the *Attribute* argument is returned. The type of value that is returned depends on what the *Attribute* argument specifies.
SQLINTEGER	*BufferLength*	input	Specifies the maximum size of buffer to which the *ValuePtr* argument points. The following conditions apply to this argument: • If *ValuePtr* points to a character string, this argument should specify the length, in bytes, of the buffer or the value SQL_NTS for nul-terminated strings. If you specify SQL_NTS, the driver assumes that the length of the string that is returned is SQL_MAX_OPTIONS_STRING_LENGTH bytes (excluding the nul-terminator). • If *ValuePtr* points to an integer, the *BufferLength* argument is ignored.
SQLINTEGER *	*StringLengthPtr*	output	Points to a buffer in which the total number of bytes (excluding the number of bytes returned for the nul-termination character) that are associated with the *ValuePtr* argument. 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 or equal to *BufferLength*, the data in *ValuePtr* is truncated to *BufferLength* minus the length of a nul-termination character. DB2 ODBC then nul-terminates this value. If the *Attribute* argument does not denote a string, then DB2 ODBC ignores the *BufferLength* argument and does not set a value in the buffer to which *StringLengthPtr* points.

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, see Table 199 on page 361.

Return codes

After you call SQLGetEnvAttr(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 112 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 112. SQLGetEnvAttr() SQLSTATEs

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory that is required to support execution or completion of the function.
HY092	Option type out of range.	An invalid value for the *Attribute* argument is specified.

Restrictions

None.

Example

The following example prints the current value of an environment attribute. SQLGetEnvAttr() retrieves the current value of the attribute SQL_ATTR_OUTPUT_NTS.

```
SQLINTEGER output_nts,autocommit;
rc = SQLGetEnvAttr(henv, SQL_ATTR_OUTPUT_NTS, &output_nts, 0, 0);
CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc );
printf("\nNull Termination of Output strings is: ");
if (output_nts == SQL_TRUE)
 printf("True\n");
else
 printf("False\n");
```

Related functions

The following functions relate to SQLGetEnvAttr() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetEnvAttr() in your applications.
- "SQLAllocHandle() - Allocate a handle" on page 72
- "SQLSetEnvAttr() - Set environment attribute" on page 360

SQLGetFunctions() - Get functions

Purpose

Table 113. SQLGetFunctions() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

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          hdbc,
                              SQLUSMALLINT     fFunction,
                              SQLUSMALLINT FAR *pfExists);
```

Function arguments

Table 114 lists the data type, use, and description for each argument in this function.

Table 114. SQLGetFunctions() arguments

Data type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Specifies a database connection handle.
SQLUSMALLINT	*fFunction*	input	Specifies which function is queried. Table 115 shows valid *fFunction* values.
SQLUSMALLINT *	*pfExists*	output	Points to the buffer where this function returns SQL_TRUE or SQL_FALSE. If the function that is queried is supported, SQL_TRUE is returned into the buffer. If the function is not supported, SQL_FALSE is returned into the buffer.

Usage

Table 115 shows the valid values for the *fFunction* argument and whether the corresponding function is supported.

If the *fFunction* argument is set to SQL_API_ALL_FUNCTIONS, then the *pfExists* argument must point to an SQLSMALLINT array of 100 elements. The array is indexed by the values in the *fFunction* argument that are used to identify many of the functions. Some elements of the array are unused and reserved. Because some values for the *fFunction* argument are greater than 100, the array method can not be used to obtain a list of all functions. The SQLGetFunctions() call must be explicitly issued for all values equal to or above 100 for the *fFunction* argument. The complete set of *fFunction* values is defined in sqlcli1.h.

Table 115. SQLGetFunctions() functions and values

fFunction	Value DB2 ODBC returns
SQL_API_SQLALLOCCONNECT	SQL_TRUE
SQL_API_SQLALLOCENV	SQL_TRUE

Table 115. SQLGetFunctions() functions and values (continued)

fFunction	**Value DB2 ODBC returns**
SQL_API_SQLALLOCHANDLE	SQL_TRUE
SQL_API_SQLALLOCSTMT	SQL_TRUE
SQL_API_SQLBINDCOL	SQL_TRUE
SQL_API_SQLBINDFILETOCOL	SQL_FALSE
SQL_API_SQLBINDFILETOPARAM	SQL_FALSE
SQL_API_SQLBINDPARAMETER	SQL_TRUE
SQL_API_SQLBROWSECONNECT	SQL_FALSE
SQL_API_SQLCANCEL	SQL_TRUE
SQL_API_SQLCLOSECURSOR	SQL_TRUE
SQL_API_SQLCOLATTRIBUTE	SQL_TRUE
SQL_API_SQLCOLATTRIBUTES	SQL_TRUE
SQL_API_SQLCOLUMNPRIVILEGES	SQL_TRUE
SQL_API_SQLCOLUMNS	SQL_TRUE
SQL_API_SQLCONNECT	SQL_TRUE
SQL_API_SQLDATASOURCES	SQL_TRUE
SQL_API_SQLDESCRIBECOL	SQL_TRUE
SQL_API_SQLDESCRIBEPARAM	SQL_TRUE
SQL_API_SQLDISCONNECT	SQL_TRUE
SQL_API_SQLDRIVERCONNECT	SQL_TRUE
SQL_API_SQLENDTRAN	SQL_TRUE
SQL_API_SQLERROR	SQL_TRUE
SQL_API_SQLEXECDIRECT	SQL_TRUE
SQL_API_SQLEXECUTE	SQL_TRUE
SQL_API_SQLEXTENDEDFETCH	SQL_TRUE
SQL_API_SQLFETCH	SQL_TRUE
SQL_API_SQLFOREIGNKEYS	SQL_TRUE
SQL_API_SQLFREECONNECT	SQL_TRUE
SQL_API_SQLFREEENV	SQL_TRUE
SQL_API_SQLFREEHANDLE	SQL_TRUE
SQL_API_SQLFREESTMT	SQL_TRUE
SQL_API_SQLGETCONNECTATTR	SQL_TRUE
SQL_API_SQLGETCONNECTOPTION	SQL_TRUE
SQL_API_SQLGETCURSORNAME	SQL_TRUE
SQL_API_SQLGETDATA	SQL_TRUE
SQL_API_SQLGETDIAGREC	SQL_TRUE
SQL_API_SQLGETENVATTR	SQL_TRUE
SQL_API_SQLGETFUNCTIONS	SQL_TRUE
SQL_API_SQLGETINFO	SQL_TRUE
SQL_API_SQLGETLENGTH	SQL_TRUE
SQL_API_SQLGETPOSITION	SQL_TRUE

Table 115. SQLGetFunctions() functions and values (continued)

fFunction	**Value DB2 ODBC returns**
SQL_API_SQLGETSQLCA	SQL_TRUE
SQL_API_SQLGETSTMTATTR	SQL_TRUE
SQL_API_SQLGETSTMTOPTION	SQL_TRUE
SQL_API_SQLGETSUBSTRING	SQL_TRUE
SQL_API_SQLGETTYPEINFO	SQL_TRUE
SQL_API_SQLMORERESULTS	SQL_TRUE
SQL_API_SQLNATIVESQL	SQL_TRUE
SQL_API_SQLNUMPARAMS	SQL_TRUE
SQL_API_SQLNUMRESULTCOLS	SQL_TRUE
SQL_API_SQLPARAMDATA	SQL_TRUE
SQL_API_SQLPARAMOPTIONS	SQL_TRUE
SQL_API_SQLPREPARE	SQL_TRUE
SQL_API_SQLPRIMARYKEYS	SQL_TRUE
SQL_API_SQLPROCEDURECOLUMNS	SQL_TRUE
SQL_API_SQLPROCEDURES	SQL_TRUE
SQL_API_SQLPUTDATA	SQL_TRUE
SQL_API_SQLROWCOUNT	SQL_TRUE
SQL_API_SQLSETCOLATTRIBUTES	SQL_TRUE
SQL_API_SQLSETCONNECTATTR	SQL_TRUE
SQL_API_SQLSETCONNECTION	SQL_TRUE
SQL_API_SQLSETCONNECTOPTION	SQL_TRUE
SQL_API_SQLSETCURSORNAME	SQL_TRUE
SQL_API_SQLSETENVATTR	SQL_TRUE
SQL_API_SQLSETPARAM	SQL_TRUE
SQL_API_SQLSETPOS	SQL_FALSE
SQL_API_SQLSETSCROLLOPTIONS	SQL_FALSE
SQL_API_SQLSETSTMTATTR	SQL_TRUE
SQL_API_SQLSETSTMTOPTION	SQL_TRUE
SQL_API_SQLSPECIALCOLUMNS	SQL_TRUE
SQL_API_SQLSTATISTICS	SQL_TRUE
SQL_API_SQLTABLEPRIVILEGES	SQL_TRUE
SQL_API_SQLTABLES	SQL_TRUE
SQL_API_SQLTRANSACT	SQL_TRUE

Return codes

After you call SQLGetFunctions(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 116 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 116. SQLGetFunctions() SQLSTATEs

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	The argument *pfExists* specifies a null pointer.
HY010	Function sequence error.	SQLGetFunctions() is called before a database connection is established.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.

Restrictions

None.

Example

Figure 20 on page 230 shows an application that connects to a database server and checks for API support using SQLGetFunctions().

```
 /******************************************************************/
 /*  Execute SQLGetFunctions to verify that APIs required          */
 /*  by application are supported.                                 */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

typedef struct odbc_api
{
  SQLUSMALLINT   api;
  char           api_name _40];
} ODBC_API;

 /******************************************************************/
 /* CLI APIs required by application                               */
 /******************************************************************/

ODBC_API o_api [7] = {
   { SQL_API_SQLBINDPARAMETER, "SQLBindParameter" } ,
   { SQL_API_SQLDISCONNECT   , "SQLDisconnect"    } ,
   { SQL_API_SQLGETTYPEINFO  , "SQLGetTypeInfo"   } ,
   { SQL_API_SQLFETCH        , "SQLFetch"         } ,
   { SQL_API_SQLTRANSACT     , "SQLTransact"      } ,
   { SQL_API_SQLBINDCOL      , "SQLBindCol"       } ,
   { SQL_API_SQLEXECDIRECT   , "SQLExecDirect"    }
                                        } ;
```

Figure 20. An application that checks the database server for API support (Part 1 of 3)

```
  /*****************************************************************/
  /* Validate that required APIs are supported.                    */
  /*****************************************************************/

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   int             i;

   // SQLGetFunctions parameters

   SQLUSMALLINT     fExists  = SQL_TRUE;
   SQLUSMALLINT    *pfExists = &fExists;


   (void) printf ("**** Entering CLIP05.\n\n");

  /*****************************************************************/
  /* Allocate environment handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate connection handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a connect handle
     goto dberror;

  /*****************************************************************/
  /* CONNECT TO data source (STLEC1)                               */
  /*****************************************************************/

   RETCODE = SQLConnect(hDbc,        // Connect handle
                     (SQLCHAR *) "STLEC1", // DSN
                     SQL_NTS,     // DSN is nul-terminated
                     NULL,        // Null UID
                     0   ,
                     NULL,        // Null Auth string
                     0);

   if( RETCODE != SQL_SUCCESS )      // Connect failed
     goto dberror;
```

Figure 20. An application that checks the database server for API support (Part 2 of 3)

```
  /******************************************************************/
  /* See if DSN supports required ODBC APIs                         */
  /******************************************************************/

   for (i = 0, (*pfExists = SQL_TRUE);
        (i < (sizeof(o_api)/sizeof(ODBC_API)) && (*pfExists) == SQL_TRUE);
        i++)
   {
     RETCODE = SQLGetFunctions (hDbc,
                                o_api[i].api,
                                pfExists);

     if (*pfExists == SQL_TRUE)      // if api is supported then print
     {
       (void) printf ("**** ODBC api %s IS supported.\n",
                      o_api[i].api_name);
     }
   }

   if (*pfExists == SQL_FALSE)      // a required api is not supported
   {
     (void) printf ("**** ODBC api %s not supported.\n",
                    o_api[i].api_name);
   }

  /******************************************************************/
  /* DISCONNECT from data source                                    */
  /******************************************************************/

   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /******************************************************************/
  /* Deallocate connection handle                                   */
  /******************************************************************/

   RETCODE = SQLFreeHandle(SQL_HANDLE_DBC, hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /******************************************************************/
  /* Free environment handle                                        */
  /******************************************************************/

   RETCODE = SQLFreeHandle(SQL_HANDLE_ENV, hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf("\n\n**** Exiting  CLIP05.\n\n   ");

   return(RETCODE);
}
```

Figure 20. An application that checks the database server for API support (Part 3 of 3)

Related functions

No functions directly relate to SQLGetFunctions().

SQLGetInfo() - Get general information

Purpose

Table 117. SQLGetInfo() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLGetInfo() returns general information (including supported data conversions) about the DBMS to which the application is currently connected.

Syntax

```
SQLRETURN SQLGetInfo (SQLHDBC       ConnectionHandle,
                      SQLUSMALLINT  InfoType,
                      SQLPOINTER    InfoValuePtr,
                      SQLSMALLINT   BufferLength,
                      SQLSMALLINT   *FAR StringLengthPtr);
```

Function arguments

Table 118 lists the data type, use, and description for each argument in this function.

Table 118. SQLGetInfo() arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Specifies a connection handle
SQLUSMALLINT	*InfoType*	input	Specifies the type of information to request. This argument must be one of the values in the first column of Table 119 on page 235.
SQLPOINTER	*InfoValuePtr*	output (and input)	Points to a buffer where this function stores the retrieved information. Depending on the type of information that is retrieved, one of the following 5 types of information is returned: • 16-bit integer value • 32-bit integer value • 32-bit binary value • 32-bit mask • Nul-terminated character string
SQLSMALLINT	*BufferLength*	input	Specifies the maximum length, in bytes, of the buffer to which the *InfoValuePtr* argument points.
SQLSMALLINT *	*StringLengthPtr*	output	Points to the buffer where this function returns the number of bytes that are required to avoid truncation of the output information. In the case of string output, this size does not include the nul-terminator. If the value in the location pointed to by *StringLengthPtr* is greater than the size of the *InfoValuePtr* buffer as specified in *BufferLength*, the string output information is truncated to *BufferLength* - 1 bytes and the function returns with SQL_SUCCESS_WITH_INFO.

Usage

Table 119 lists the possible values for the *InfoType* argument and a description of the information that SQLGetInfo() returns for each value. (This table indicates which *InfoType* argument values were renamed in ODBC 3.0. For those *InfoType* argument values that were renamed, Table 120 on page 255 lists the ODBC 2.0 and 3.0 names.)

Important: If the value that is specified for the *InfoType* argument does not apply or is not supported, the result is dependent on the return type. The following values are returned for each type of unsupported value in the *InfoType* argument:
- 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).

Table 119 specifies each value that you can specify for the *InfoType* argument and describes the information that each of these values will return.

Table 119. Information returned by SQLGetInfo()

InfoType	**Format**	**Description and notes**
SQL_ACCESSIBLE_PROCEDURES	string	A character string of 'Y' indicates that the user can execute all procedures returned by the function SQLProcedures(). 'N' indicates that procedures can be 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 tables can be returned that the user cannot access.
SQL_ACTIVE_ENVIRONMENTS	16-bit integer	The maximum number of active environments that the DB2 ODBC driver can support. If the limit is unspecified or unknown, this value is set to zero.
SQL_AGGREGATE_FUNCTIONS	32-bit mask	A bit mask 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 ODBC returns 0 indicating that the ALTER DOMAIN statement is not supported. ODBC also defines the following values that DB2 ODBC does not return: • 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

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	Format	Description and notes
SQL_ALTER_TABLE	32-bit mask	Indicates which clauses in ALTER TABLE are supported by the DBMS. • SQL_AT_ADD_COLUMN • SQL_AT_DROP_COLUMN
SQL_ASCII_GCCSID	32-bit integer	Specifies the ASCII GCCSID value currently set in the AGCCSID field of DB2 DSNHDECP.
SQL_ASCII_MCCSID	32-bit integer	Specifies the ASCII MCCSID value currently set in the AMCCSID field of DB2 DSNHDECP.
SQL_ASCII_SCCSID	32-bit integer	Specifies the ASCII SCCSID value currently set in the ASCCSID field of DB2 DSNHDECP.
SQL_BATCH_ROW_COUNT	32-bit mask	Indicates the availability of row counts. DB2 ODBC 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 DB2 ODBC does not return: • 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	Reserved attribute, zero is returned for the bit-mask.
SQL_CATALOG_LOCATION (In previous versions of DB2 ODBC, this *InfoType* is SQL_QUALIFIER_LOCATION.)	16-bit integer	A 16-bit integer value indicated the position of the qualifier in a qualified table name. Zero indicates that qualified names are not supported.
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 (In previous versions of DB2 ODBC, this *InfoType* is SQL_QUALIFIER_NAME_SEPARATOR.)	string	The characters used as a separator between a catalog name and the qualified name element that follows it.
SQL_CATALOG_TERM (In previous versions of DB2 ODBC, this *InfoType* is 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. Because DB2 ODBC does not support three part names, a zero-length string is returned. For non-ODBC applications, the SQL_CATALOG_TERM symbolic name should be used instead of SQL_QUALIFIER_NAME.
SQL_CATALOG_USAGE (In previous versions of DB2 ODBC, this *InfoType* is SQL_QUALIFIER_USAGE.)	32-bit mask	This is similar to SQL_OWNER_USAGE except that this is used for catalog.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
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, EBCDIC). If this is unknown, an empty string is 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_DAY_TIME SQL_CONVERT_INTERVAL_YEAR_MONTH SQL_CONVERT_LONGVARBINARY SQL_CONVERT_LONGVARCHAR SQL_CONVERT_NUMERIC SQL_CONVERT_REAL SQL_CONVERT_ROWID SQL_CONVERT_SMALLINT SQL_CONVERT_TIME SQL_CONVERT_TIMESTAMP SQL_CONVERT_TINYINT SQL_CONVERT_VARBINARY SQL_CONVERT_VARCHAR	32-bit mask	Indicates the conversions supported by the data source with the CONVERT scalar function for data of the type named in the *InfoType*. If the bit mask 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 bit mask with SQL_CVT_DECIMAL. If the resulting value is nonzero then the conversion is supported. The following bit masks 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_LONGVARBINARY • SQL_CVT_LONGVARCHAR • SQL_CVT_NUMERIC • SQL_CVT_REAL • SQL_CVT_ROWID • 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. • SQL_FN_CVT_CONVERT - used to determine which conversion functions are supported. • SQL_FN_CVT_CAST - used to determine which cast functions are supported.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
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 represents.
SQL_CLOSE_BEHAVIOR	32-bit integer	Indicates whether 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 *fOption* set to SQL_CLOSE or the statement handle is freed with SQLFreeHandle(). In addition, the end of the transaction (when a commit or rollback is issued) can also cause the closing of the cursor (depending on the WITH HOLD attribute currently in use).
SQL_CREATE_ASSERTION	32-bit mask	Indicates which clauses in the CREATE ASSERTION statement are supported by the DBMS. DB2 ODBC always returns zero; the CREATE ASSERTION statement is not supported. ODBC also defines the following values that DB2 ODBC does not return: • 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 ODBC always returns zero; the CREATE CHARACTER SET statement is not supported. ODBC also defines the following values that DB2 ODBC does not return: • 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 COLLATION statement are supported by the DBMS. DB2 ODBC always returns zero; the CREATE COLLATION statement is not supported. ODBC also defines the following values that DB2 ODBC does not return: • SQL_CCOL_CREATE_COLLATION
SQL_CREATE_DOMAIN	32-bit mask	Indicates which clauses in the CREATE DOMAIN statement are supported by the DBMS. DB2 ODBC always returns zero; the CREATE DOMAIN statement is not supported. ODBC also defines the following values that DB2 ODBC does not return: • 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

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
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 bit masks 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 ODBC always returns zero; the CREATE TRANSLATION statement is not supported. ODBC also defines the following value that DB2 ODBC does not return: • SQL_CTR_CREATE_TRANSLATION

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
SQL_CURSOR_COMMIT_BEHAVIOR	16-bit integer	Indicates how a COMMIT operation affects cursors. A value of: • SQL_CB_DELETE, destroys cursors and drops access plans for dynamic SQL statements. • SQL_CB_CLOSE, destroys 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. 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, destroys cursors and drops access plans for dynamic SQL statements. • SQL_CB_CLOSE, destroys 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. 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, it 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; an ″N″ indicates that it is not set to READ ONLY mode.
SQL_DATABASE_NAME	string	The name of the current database in use. Also, this information returned by SELECT CURRENT SERVER on IBM DBMSs.
SQL_DBMS_NAME	string	The name of the DBMS product being accessed. For example: • ″DB2/6000″ • ″DB2/2″

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
SQL_DBMS_VER	string	The version of the DBMS product being 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, ″02.01.0000″ translates to major version 2, 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 the IBM 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 the IBM 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 the IBM 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 the IBM RR level. • SQL_TXN_VERSIONING = Not applicable to IBM DBMSs. • SQL_TXN_NOCOMMIT = Any changes are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed. This is a DB2 UDB for iSeries 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_DRIVER_HDBC	32 bits	DB2 ODBC’s current database handle.
SQL_DRIVER_HENV	32 bits	DB2 ODBC’s environment handle.
SQL_DRIVER_HLIB	32 bits	Reserved.
SQL_DRIVER_HSTMT	32 bits	DB2 ODBC’s current statement handle for the current connection.
SQL_DRIVER_NAME	string	The file name of the DB2 ODBC implementation. DB2 ODBC returns NULL.
SQL_DRIVER_ODBC_VER	string	The version number of ODBC that the driver supports. DB2 ODBC returns ″3.00″.
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, ″03.01.0000″ translates to major version 3, minor version 1, release 0.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
SQL_DROP_ASSERTION	32-bit mask	Indicates which clause in the DROP ASSERTION statement is supported by the DBMS. DB2 ODBC always returns zero; the DROP ASSERTION statement is not supported. ODBC also defines the following value that DB2 ODBC does not return: • SQL_DA_DROP_ASSERTION
SQL_DROP_CHARACTER_SET	32-bit mask	Indicates which clause in the DROP CHARACTER SET statement is supported by the DBMS. DB2 ODBC always returns zero; the DROP CHARACTER SET statement is not supported. ODBC also defines the following value that DB2 ODBC does not return. • SQL_DCS_DROP_CHARACTER_SET
SQL_DROP_COLLATION	32-bit mask	Indicates which clause in the DROP COLLATION statement is supported by the DBMS. DB2 ODBC always returns zero; the DROP COLLATION statement is not supported. ODBC also defines the following value that DB2 ODBC does not return: • SQL_DC_DROP_COLLATION
SQL_DROP_DOMAIN	32-bit mask	Indicates which clauses in the DROP DOMAIN statement are supported by the DBMS. DB2 ODBC always returns zero; the DROP DOMAIN statement is not supported. ODBC also defines the following values that DB2 ODBC does not return: • SQL_DD_DROP_DOMAIN • SQL_DD_CASCADE • SQL_DD_RESTRICT
SQL_DROP_SCHEMA	32-bit mask	Indicates which clauses in the DROP SCHEMA statement are supported by the DBMS. • SQL_DS_DROP_SCHEMA • SQL_DS_CASCADE • SQL_DS_RESTRICT
SQL_DROP_TABLE	32-bit mask	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 mask	Indicates which clauses in the DROP TRANSLATION statement are supported by the DBMS. DB2 ODBC always returns zero; the DROP TRANSLATION statement is not supported. ODBC also defines the following value that DB2 ODBC does not return: • SQL_DTR_DROP_TRANSLATION
SQL_DROP_VIEW	32-bit mask	Indicates which clauses in the DROP VIEW statement are supported by the DBMS. • SQL_DV_DROP_VIEW • SQL_DV_CASCADE • SQL_DV_RESTRICT

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
SQL_DYNAMIC_CURSOR_ATTRIBUTES1	32-bit mask	Indicates the attributes of a dynamic cursor that DB2 ODBC supports (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 DB2 ODBC supports (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_EBCDIC_GCCSID	32-bit integer	Specifies the EBCDIC GCCSID value currently set in the AGCCSID field of DB2 DSNHDECP.
SQL_EBCDIC_MCCSID	32-bit integer	Specifies the EBCDIC MCCSID value currently set in the AMCCSID field of DB2 DSNHDECP.
SQL_EBCDIC_SCCSID	32-bit integer	Specifies the EBCDIC SCCSID value currently set in the ASCCSID field of DB2 DSNHDECP.
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 attribute values 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	Reserved. Zero is returned.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1	32-bit mask	Indicates the attributes of a forward-only cursor that DB2 ODBC supports (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 DB2 ODBC supports (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 ODBC: • 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 SQL_GD_BLOCK and SQL_GD_BOUND; these bits are not returned by DB2 ODBC.
SQL_GROUP_BY	16-bit integer	Indicates the degree of support for the GROUP BY clause by the server: • SQL_GB_NO_RELATION, the columns in the GROUP BY and in the SELECT list are not related • 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

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
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. **IBM specific:** Identifier names in IBM DBMSs are not case sensitive.
SQL_IDENTIFIER_QUOTE_CHAR	string	Indicates the character used to surround a delimited identifier.
SQL_INFO_SCHEMA_VIEWS	32-bit mask	Indicates the views in the INFORMATIONAL_SCHEMA that are supported. DB2 ODBC always returns zero; no views in the INFORMATIONAL_SCHEMA are supported. ODBC also defines the following values that DB2 ODBC does not return: • 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 (In previous versions of DB2 ODBC, this *InfoType* is SQL_ODBC_SQL_OPT_IEF.)	string	A 'Y' indicates that the data source supports Integrity Enhanced Facility (IEF) in SQL89 and in X/Open XPG4 Embedded SQL; an 'N' indicates that it does not.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	Format	Description and notes
SQL_KEYSET_CURSOR_ATTRIBUTES1	32-bit mask	Indicates the attributes of a keyset cursor that DB2 ODBC supports (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 DB2 ODBC supports (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	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 attribute, 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 ODBC can support on a given connection. This value is zero if this number has no specific limit, or the limit is unknown.
SQL_MAX_BINARY_LITERAL_LEN	32-bit integer	A 32-bit integer value specifying the maximum length of a hexadecimal literal in a SQL statement.
SQL_MAX_CATALOG_NAME_LEN (In previous versions of DB2 ODBC, this *InfoType* is SQL_MAX_QUALIFIER_NAME_LEN.)	16-bit integer	The maximum length of a catalog qualifier name; first part of a three-part table name (in bytes).
SQL_MAX_CHAR_LITERAL_LEN	32-bit 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.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
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 (In previous versions of DB2 ODBC, this *InfoType* is SQL_ACTIVE_STATEMENTS.)	16-bit integer	The maximum number of active statements per connection. Zero is returned, indicating that the limit is dependent on database system and DB2 ODBC resources, and limits.
SQL_MAX_CURSOR_NAME_LEN	16-bit integer	The maximum length of a cursor name (in bytes).
SQL_MAX_DRIVER_CONNECTIONS (In previous versions of DB2 ODBC, this *InfoType* is SQL_ACTIVE_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 initialization file or the SQL_MAX_CONNECTIONS environment and connection attribute 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_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 integer	Indicates the maximum size in bytes that the server supports for the combined columns in an index. Zero if no limit.
SQL_MAX_PROCEDURE_NAME_LEN	16-bit integer	The maximum length of a procedure name (in bytes).
SQL_MAX_ROW_SIZE	32-bit 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	Returns 'Y' if SQLGetInfo() with *InfoType* set to SQL_MAX_ROW_SIZE includes the length of product-specific *long string* data types. Otherwise, returns 'N'.
SQL_MAX_SCHEMA_NAME_LEN (In previous versions of DB2 ODBC, this *InfoType* is SQL_MAX_OWNER_NAME_LEN.)	16-bit integer	The maximum length of a schema qualifier name (in bytes).
SQL_MAX_STATEMENT_LEN	32-bit integer	Indicates the maximum length, in bytes, of an SQL statement string, which includes 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.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
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.
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 null values 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 465. 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_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.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
SQL_ODBC_SQL_CONFORMANCE	16-bit integer	A value of: • SQL_OSC_MINIMUM - means that the current DBMS supports minimum ODBC SQL grammar. Minimum SQL grammar must include the following elements: – CREATE TABLE and DROP TABLE data definitions – Simple SELECT, INSERT, UPDATE, and DELETE data manipulation – Simple expressions – CHAR, VARCHAR, and LONG VARCHAR data types • SQL_OSC_CORE - means that the current DBMS supports ODBC SQL core grammar. Core ODBC SQL grammar must include the following elements: – Minimum ODBC SQL grammar – ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW, GRANT, and REVOKE data definitions – Full SELECT data manipulation – Subquery and function expressions – DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE PRECISION data types • SQL_OSC_EXTENDED - means the current DBMS supports extended ODBC SQL grammar. Extended ODBC SQL must grammar include the following elements: – Core ODBC SQL grammar – Positioned UPDATE, positioned DELETE, SELECT FOR UPDATE, and UNION data definitions – Scalar functions, literal date, literal time, and literal timestamp expressions – BIT, TINYINT, BIGINT, BINARY, VARBINARY, LONG VARBINARY, DATE, TIME, TIMESTAMP data types – Batch SQL statements – Procedure calls
SQL_ODBC_VER	string	The version number of ODBC that the driver manager supports. DB2 ODBC returns the string ″03.01.0000″.
SQL_OJ_CAPABILITIES	32-bit mask	A 32-bit bit mask enumerating the types of outer join supported. The bit masks 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_NOT_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_OPS: Any predicate can 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’.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
SQL_OUTER_JOINS	string	The character string: • 'Y' indicates that outer joins are supported, and DB2 ODBC supports the ODBC outer join request syntax. • 'N' indicates that it is not supported. (See "Using vendor escape clauses" on page 465)
SQL_OWNER_TERM (In previous versions of DB2 ODBC, this *InfoType* is SQL_SCHEMA_TERM.)	string	The database vendor's (owner's) terminology for a schema
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: Only one row count is 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: One result set is 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: Only one result set is 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 attribute, 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 the server requires the FOR UPDATE clause to be specified on a <query expression> in order for a column to be updatable using the cursor.
SQL_PROCEDURE_TERM	string	The name a database vendor uses for a procedure
SQL_PROCEDURES	string	'Y' indicates that the data source supports procedures and DB2 ODBC supports the ODBC procedure invocation syntax specified in "Using stored procedures" on page 429. 'N' indicates that it does not.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
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	'Y' indicates that changes are detected in rows between multiple fetches of the same rows. 'N' indicates that changes are not detected.
SQL_SCHEMA_USAGE (In previous versions of DB2 ODBC, this *InfoType* is SQL_OWNER_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_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 (for example, grant and revoke statements).
SQL_SCROLL_CONCURRENCY	32-bit mask	Indicates the concurrency attribute values supported for the cursor. The following bit-masks are used in conjunction with the flag to determine which attribute values are supported: • SQL_SCCO_READ_ONLY • SQL_SCCO_LOCK • SQL_SCCO_OPT_TIMESTAMP • SQL_SCCO_OPT_VALUES DB2 ODBC returns SQL_SCCO_LOCK, indicating that the lowest level of locking that is sufficient to ensure that the row can be updated is used.
SQL_SCROLL_OPTIONS	32-bit mask	The scroll attribute values supported for scrollable cursors. The following bit masks are used in conjunction with the flag to determine which attribute values are supported: • SQL_SO_FORWARD_ONLY • SQL_SO_KEYSET_DRIVEN • SQL_SO_STATIC • SQL_SO_DYNAMIC • SQL_SO_MIXED DB2 ODBC returns SQL_SO_FORWARD_ONLY, indicating that the cursor scrolls forward only.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
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 subsystem to which the application is connected.
SQL_SPECIAL_CHARACTERS	string	Contains all the characters that the server allows in non-delimited identifiers. This includes a...z, A...Z, 0...9, and _.
SQL_SQL92_PREDICATES	32-bit mask	Indicates those predicates that are defined by SQL92 and that are supported in a SELECT statement. • 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_VALUE_EXPRESSIONS	32-bit mask	Indicates those value expressions that are defined by SQL92 and that are supported. • SQL_SVE_CASE • SQL_SVE_CAST • SQL_SVE_COALESCE • SQL_SVE_NULLIF
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.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	**Format**	**Description and notes**
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_CHAR • 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_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 *string1, string2,* and *start* arguments, the SQL_FN_STR_LOCATE bit mask is returned. If an application can only call the LOCATE scalar function with the *string1* and *string2*, the SQL_FN_STR_LOCATE_2 bit mask is returned. If the LOCATE scalar function is fully supported, both bit masks are returned.
SQL_SUBQUERIES	32-bit mask	Indicates which predicates support subqueries: • SQL_SQ_COMPARISON - 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 **Tip:** 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.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	Format	Description and notes
SQL_TIMEDATE_ADD_INTERVALS	32-bit mask	Indicates whether the special ODBC system function TIMESTAMPADD is supported, and, if it is, which intervals are supported. The following bit masks 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 the special ODBC system function TIMESTAMPDIFF is supported, and, if it is, which intervals are supported. The following bit masks 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_CURDATE • SQL_FN_TD_CURTIME • SQL_FN_TD_DAYNAME • SQL_FN_TD_DAYOFMONTH • SQL_FN_TD_DAYOFWEEK • SQL_FN_TD_DAYOFYEAR • 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 **Tip:** These functions are intended to be used with the escape sequence in ODBC.

Table 119. Information returned by SQLGetInfo() (continued)

InfoType	Format	Description and notes
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, and so on) DDL statements (CREATE TABLE, DROP INDEX, and so on) 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 bit masks are used in conjunction with the flag to determine which attribute values 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, see SQL_DEFAULT_TXN_ISOLATION.
| SQL_UNICODE_GCCSID	32-bit integer	Specifies the UNICODE GCCSID value currently set in the UGCCSID field of DB2 DSNHDECP.
| SQL_UNICODE_MCCSID	32-bit integer	Specifies the UNICODE MCCSID value currently set in the UMCCSID field of DB2 DSNHDECP.
| SQL_UNICODE_SCCSID	32-bit integer	Specifies the UNICODE SCCSID value currently set in the USCCSID field of DB2 DSNHDECP.
SQL_UNION	32-bit mask	Indicates whether 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, SQL_U_UNION is set as well.
SQL_USER_NAME	string	The user name that is 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.

Table 120 lists ODBC 2.0 *InfoType* values that are renamed in the ODBC 3.0 specification.

Table 120. Renamed SQLGetInfo() InfoTypes

ODBC 2.0 *InfoType*	**ODBC 3.0** *InfoType*
SQL_ACTIVE_CONNECTIONS	SQL_MAX_DRIVER_CONNECTIONS
SQL_ACTIVE_STATEMENTS	SQL_MAX_CONCURRENT_ACTIVITIES

Table 120. Renamed SQLGetInfo() InfoTypes (continued)

ODBC 2.0 *InfoType*	**ODBC 3.0** *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

Return codes

After you call SQLGetInfo(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 121 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 121. SQLGetInfo() SQLSTATEs

SQLSTATE	**Description**	**Explanation**
01004	Data truncated.	The requested information is returned as a string and its length exceeds the length of the application buffer as specified in the *BufferLength* argument. The *StringLengthPtr* argument contains the actual (not truncated) length, in bytes, of the requested information. (SQLGetInfo() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
08003	Connection is closed.	The type of information that the *InfoType* argument requests requires an open connection. Only the value SQL_ODBC_VER does not require an open connection.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* is less than 0.
HY096	Invalid information type.	An invalid value is specified for the *InfoType* argument.
HYC00	Driver not capable.	The value specified in the argument *InfoType* is not supported by DB2 ODBC or is not supported by the data source.

Restrictions

None.

Example

The following lines of code use SQLGetInfo() to retrieve the current data source name:

```
SQLCHAR        buffer[255];
SQLSMALLINT    outlen;

rc = SQLGetInfo(hdbc, SQL_DATA_SOURCE_NAME, buffer, 255, &outlen);
printf("\nServer Name: %s\n", buffer);
```

Related functions

The following functions relate to SQLGetInfo() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetInfo() in your applications.

- “SQLGetConnectAttr() - Get current attribute setting” on page 196
- “SQLGetTypeInfo() - Get data type information” on page 280

SQLGetLength() - Retrieve length of a string value

Purpose

Table 122. SQLGetLength() specifications

ODBC	X/OPEN CLI	ISO CLI
No	No	No

SQLGetLength() retrieves the length (in bytes) of a large object value that is referenced by a large object locator that the server returns (as a result of a fetch, or an SQLGetSubString() call) during the current transaction.

Syntax

```
SQLRETURN  SQLGetLength      (SQLHSTMT          hstmt,
                              SQLSMALLINT       LocatorCType,
                              SQLINTEGER        Locator,
                              SQLINTEGER  FAR   *StringLength,
                              SQLINTEGER  FAR   *IndicatorValue);
```

Function arguments

Table 123 lists the data type, use, and description for each argument in this function.

Table 123. SQLGetLength() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle. This can be any statement handle that is allocated but does not currently have a prepared statement assigned to it.
SQLSMALLINT	*LocatorCType*	input	Specifies the C type of the source LOB locator. This must be one of the following values: • SQL_C_BLOB_LOCATOR for BLOB data • SQL_C_CLOB_LOCATOR for CLOB data • SQL_C_DBCLOB_LOCATOR for DBCLOB data
SQLINTEGER	*Locator*	input	Specifies the LOB locator value. This argument specifies a LOB locator value not the LOB value itself.
SQLINTEGER *	*StringLength*	output	Points to a buffer that receives the length (in bytes[1]) of the LOB to which the locator argument refers.
SQLINTEGER *	*IndicatorValue*	output	This argument is always returns zero.

Note:

1. This is in bytes even for DBCLOB data.

Usage

SQLGetLength() can determine the length of the data value represented by a LOB locator. Applications use it to determine the overall length of the referenced LOB value so that the appropriate strategy for obtaining some or all of that value can be chosen.

The *Locator* argument can contain any valid LOB locator that is not explicitly freed using a FREE LOCATOR statement or that is implicitly freed because the transaction during which it was created has terminated.

The statement handle must not be associated with any prepared statements or catalog function calls.

Return codes

After you call SQLGetLength(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 124 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 124. SQLGetLength() SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The combination of the values that the *LocatorCType* and *Locator* arguments specify is not valid.
0F001	The LOB token variable does not currently represent any value.	The value that the *Locator* argument specifies is not associated with a LOB locator.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY003	Program type out of range.	The *LocatorCType* argument does not specify one of the following values: • SQL_C_CLOB_LOCATOR • SQL_C_BLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR
HY009	Invalid use of a null pointer.	The *StringLength* argument specifies a null pointer.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is 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.

Restrictions

This function is not available when you connect to a DB2 server that does not support large objects. Call SQLGetFunctions() with the *fFunction* argument set to SQL_API_SQLGETLENGTH and check the *fExists* output argument to determine if the function is supported for the current connection.

Example

See Figure 21 on page 264.

Related functions

The following functions relate to SQLGetLength() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetLength() in your applications.

- "SQLBindCol() - Bind a column to an application variable" on page 78
- "SQLExtendedFetch() - Fetch an array of rows" on page 163
- "SQLFetch() - Fetch the next row" on page 171
- "SQLGetPosition() - Find the starting position of a string" on page 261
- "SQLGetSubString() - Retrieve portion of a string value" on page 276

SQLGetPosition() - Find the starting position of a string

Purpose

Table 125. SQLGetPosition() specifications

ODBC	X/OPEN CLI	ISO CLI
No	No	No

SQLGetPosition() returns 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 value that is returned from the database from a fetch or a SQLGetSubString() call during the current transaction.

Syntax

```
SQLRETURN  SQLGetPosition    (SQLHSTMT          hstmt,
                              SQLSMALLINT       LocatorCType,
                              SQLINTEGER        SourceLocator,
                              SQLINTEGER        SearchLocator,
                              SQLCHAR     FAR   *SearchLiteral,
                              SQLINTEGER        SearchLiteralLength,
                              SQLUINTEGER       FromPosition,
                              SQLUINTEGER FAR   *LocatedAt,
                              SQLINTEGER  FAR   *IndicatorValue);
```

Function arguments

Table 126 lists the data type, use, and description for each argument in this function.

Table 126. SQLGetPosition() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle. This can be any statement handle that is allocated but does not currently have a prepared statement assigned to it.
SQLSMALLINT	*LocatorCType*	input	Specifies the C type of the source LOB locator. This argument must specify one of the following values: • SQL_C_BLOB_LOCATOR for BLOB data • SQL_C_CLOB_LOCATOR for CLOB data • SQL_C_DBCLOB_LOCATOR for DBCLOB data
SQLINTEGER	*Locator*	input	Specifies the source LOB locator.
SQLINTEGER	*SearchLocator*	input	Specifies a LOB locator that refers to a search string. This argument is ignored unless both the following conditions are met: • The *SearchLiteral* argument specifies a null pointer. • The *SearchLiteralLength* argument is set to 0.
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.

Table 126. SQLGetPosition() arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER	*SearchLiteralLength*	input	The length of the string in *SearchLiteral* (in bytes).[1] If this argument value is 0, you specify the search string with a LOB locator. (The *SearchLocator* argument specifies the search string when it is represented by a LOB locator.)
SQLUINTEGER	*FromPosition*	input	For BLOBs and CLOBs, this argument specifies the position of the byte within the source string at which the search is to start. For DBCLOBs, this argument specifies the character at which the search is to start. The start-byte or start-character is numbered 1.
SQLUINTEGER *	*LocatedAt*	output	Specifies the position at which the search string was located. For BLOBs and CLOBs, this location is the byte position. For DBCLOBs, this location is the character position. If the search string is not located this argument returns zero. If the length of the source string is zero, the value 1 is returned.
SQLINTEGER *	*IndicatorValue*	output	Always set to zero.

Note:

1. This is in bytes even for DBCLOB data.

Usage

Use SQLGetPosition() in conjunction with SQLGetSubString() to obtain a portion of a string in a random manner. To use SQLGetSubString(), you must know the location of the substring within the overall string in advance. In situations in which you want to use a search string to find the start of a substring, use SQLGetPosition().

The *Locator* and *SearchLocator* arguments (if they are used) can contain any valid LOB locator that is not explicitly freed using a FREE LOCATOR statement or that is not implicitly freed because the transaction during which it was created has terminated.

The *Locator* and *SearchLocator* arguments must specify LOB locators of the same type.

The statement handle must not be associated with any prepared statements or catalog function calls.

Return codes

After you call SQLGetPosition(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 127 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 127. SQLGetPosition() SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The combination of the value that the *LocatorCType* argument specifies with either of the LOB locator values is not valid.
0F001	The LOB token variable does not currently represent any value.	A value specified for the *Locator* or *SearchLocator* arguments is currently not a LOB locator.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
42818	The operands of an operator or function are not compatible.	The length of the search pattern is longer than 4000 bytes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	This SQLSTATE is returned for one or more of the following reasons: • The pointer that the *LocatedAt* argument specifies is null. • The argument value for the *FromPosition* argument is not greater than 0. • The *LocatorCType* argument is not one of the following values: – SQL_C_CLOB_LOCATOR – SQL_C_BLOB_LOCATOR – SQL_C_DBCLOB_LOCATOR
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value of *SearchLiteralLength* is 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.

Restrictions

This function is available only when you connect to a DB2 server that supports large objects. Call SQLGetFunctions() with the *fFunction* argument set to SQL_API_SQLGETPOSITION and check the *fExists* output argument to determine if the function is supported for the current connection.

Example

Figure 21 on page 264 shows an application that retrieves a substring from a large object. To find the where in a large object this substring begins, the application calls SQLGetPosition().

```
/* ... */
    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_INPUT, SQL_C_CHAR, SQL_CHAR,
                          7, 0, Empno.s, sizeof(Empno.s), &Empno.ind);


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

    rc = SQLExecDirect(hstmt, stmt2, SQL_NTS);
    rc = SQLBindCol(hstmt, 1, SQL_C_CLOB_LOCATOR, &ClobLoc1, 0,
                      &pcbValue);
    rc = SQLFetch(hstmt);
```

Figure 21. An application that retrieves a substring from a large object (Part 1 of 2)

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

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

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

    /* Get Starting postion */
    rc = SQLGetPosition(lhstmt, SQL_C_CLOB_LOCATOR, ClobLoc1, 0,
                    "Interests", 9, 1, &Pos1, &Ind);


    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);

    /* Print Interest section of Employee's resume */
    printf("\nEmployee #: %s\n %s\n", Empno.s, buffer);
/* ... */
```

Figure 21. An application that retrieves a substring from a large object (Part 2 of 2)

Related functions

The following functions relate to SQLGetPosition() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetPosition() in your applications.

- “SQLBindCol() - Bind a column to an application variable” on page 78
- “SQLExtendedFetch() - Fetch an array of rows” on page 163
- “SQLFetch() - Fetch the next row” on page 171
- “SQLGetFunctions() - Get functions” on page 226
- “SQLGetLength() - Retrieve length of a string value” on page 258
- “SQLGetSubString() - Retrieve portion of a string value” on page 276

SQLGetSQLCA() - Get SQLCA data structure

Purpose

Table 128. SQLGetSQLCA specifications

ODBC	X/OPEN CLI	ISO CLI
No	No	No

SQLGetSQLCA() is used to return the SQLCA (SQL communication area) associated with preparing and executing an SQL statement, fetching data, or closing a cursor. The SQLCA can return information that supplements the information obtained by using SQLGetDiagRec().

For a detailed description of the SQLCA structure, see Appendix C of *DB2 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          henv,
                              SQLHDBC          hdbc,
                              SQLHSTMT         hstmt,
                              struct sqlca FAR *pSqlca);
```

Function arguments

Table 129 lists the data type, use, and description for each argument in this function.

Table 129. SQLGetSQLCA() arguments

Data type	Argument	Use	Description
SQLHENV	*henv*	input	Specifies the environment handle.
SQLHDBC	*hdbc*	input	Specifies a connection handle.
SQLHSTMT	*hstmt*	input	Specifies a statement handle.
SQLCA *	*pqlCA*	output	Points to a buffer to receive the SQL communication area.

Usage

The handles are used in the same way as for the SQLGetDiagRec() function. To obtain the SQLCA associated with different handle types, use the following argument values:

- For an environment handle: specify a valid environment handle, set *hdbc* to SQL_NULL_HDBC and set *hstmt* and SQL_NULL_HSTMT.
- For a connection handle: specify a valid database connection handle and set *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored.
- For a statement handle: specify a valid statement handle. The *henv* and *hdbc* arguments are ignored.

If diagnostic information that one DB2 ODBC function generates is not retrieved before a function other than SQLGetDiagRec() is called on the same handle, the diagnostic information for the previous function call is lost. This information is lost regardless of whether the second DB2 ODBC function call generates diagnostic information.

If a DB2 ODBC function is called that does not result in interaction with the DBMS, then the SQLCA contains all zeroes. Meaningful information is returned in the SQLCA for the following functions:
- SQLCancel()
- SQLConnect(), SQLDisconnect()
- SQLExecDirect(), SQLExecute()
- SQLFetch()
- SQLPrepare()
- SQLEndTran()
- SQLColumns()
- SQLConnect()
- SQLGetData (if a LOB column is involved)
- SQLSetConnectAttr() (for SQL_ATTR_AUTOCOMMIT)
- SQLStatistics()
- SQLTables()
- SQLColumnPrivileges()
- SQLExtendedFetch()
- SQLForeignKeys()
- SQLMoreResults()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLTablePrivileges()

Return codes

After you call SQLGetSQLCA(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

None.

Restrictions

None.

Example

Figure 22 on page 267 shows an application that uses SQLGetSQLCA() to retrieve diagnostic information from the SQLCA.

```
 /******************************************************************/
 /*  Prepare a query and execute that query against a non-existent */
 /*  table. Then invoke SQLGetSQLCA to extract                     */
 /*  native SQLCA data structure. Note that this API is NOT        */
 /*  defined within ODBC; it is unique to IBM CLI.         */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

void print_sqlca (SQLHENV,            // prototype for print_sqlca
                  SQLHDBC,
                  SQLHSTMT);

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            *pDSN = "STLEC1";
   SWORD           cbCursor;
   SDWORD          cbValue1;
   SDWORD          cbValue2;
   char            employee [30];
   int             salary = 0;
   int             param_salary = 30000;

   char            *stmt = "SELECT NAME, SALARY FROM EMPLOYEES WHERE SALARY > ?";


   (void) printf ("**** Entering CLIP11.\n\n");

  /*****************************************************************/
  /* Allocate environment handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate connection handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;
```

Figure 22. An application that retrieves diagnostic information (Part 1 of 5)

```
/******************************************************************/
/* CONNECT TO data source (STLEC1)                                */
/******************************************************************/

 RETCODE = SQLConnect(hDbc,        // Connect handle
                     (SQLCHAR *) pDSN, // DSN
                     SQL_NTS,      // DSN is nul-terminated
                     NULL,         // Null UID
                     0   ,
                     NULL,         // Null Auth string
                     0);

 if( RETCODE != SQL_SUCCESS )      // Connect failed
   goto dberror;

/******************************************************************/
/* Allocate statement handles                                     */
/******************************************************************/

rc = SQLAllocHandle(SQL_HANDLE_STMT, SQL_NULL_HANDLE, hDbc, &hStmt);

if (rc != SQL_SUCCESS)
  goto exit;

/******************************************************************/
/* Prepare the query for multiple execution within current       */
/* transaction. Note that query is collapsed when transaction    */
/* is committed or rolled back.                                   */
/******************************************************************/

rc = SQLPrepare (hStmt,
                 (SQLCHAR *) stmt,
                 strlen(stmt));

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** PREPARE OF QUERY FAILED.\n");
  (void) print_sqlca (hStmt,
                      hDbc,
                      hEnv);
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind employee name
                 1,
                 SQL_C_CHAR,
                 employee,
                 sizeof(employee),
                 &cbValue1);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF NAME FAILED.\n");
  goto dberror;
}
```

Figure 22. An application that retrieves diagnostic information (Part 2 of 5)

```
rc = SQLBindCol (hStmt,           // bind employee salary
                 2,
                 SQL_C_LONG,
                 &salary,
                 0,
                 &cbValue2);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF SALARY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Bind parameter to replace '?' in query. This has an initial   */
/* value of 30000.                                               */
/*****************************************************************/

rc = SQLBindParameter (hStmt,
                       1,
                       SQL_PARAM_INPUT,
                       SQL_C_LONG,
                       SQL_INTEGER,
                       0,
                       0,
                       &param_salary,
                       0,
                       NULL);

/*****************************************************************/
/* Execute prepared statement to generate answer set.            */
/*****************************************************************/

rc = SQLExecute (hStmt);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** EXECUTE OF QUERY FAILED.\n");
  (void) print_sqlca (hStmt,
                      hDbc,
                      hEnv);
  goto dberror;
}

/*****************************************************************/
/* Answer set is available -- Fetch rows and print employees     */
/* and salary.                                                   */
/*****************************************************************/

(void) printf ("**** Employees whose salary exceeds %d follow.\n\n",
               param_salary);

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Employee Name %s with salary %d.\n",
                 employee,
                 salary);
}
```

Figure 22. An application that retrieves diagnostic information (Part 3 of 5)

```
  /*****************************************************************/
  /* Deallocate statement handles -- statement is no longer in a   */
  /* Prepared state.                                               */
  /*****************************************************************/

  rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);

  /*****************************************************************/
  /* DISCONNECT from data source                                   */
  /*****************************************************************/

   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Deallocate connection handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeHandle(SQL_HANDLE_DBC, hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Free environment handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeHandle(SQL_HANDLE_ENV, hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP11.\n\n");

   return RETCODE;
}

/*****************************************************************/
/* print_sqlca invokes SQLGetSQLCA and prints the native SQLCA.  */
/*****************************************************************/

void print_sqlca (SQLHENV  hEnv ,
                  SQLHDBC  hDbc ,
                  SQLHSTMT hStmt)
{
  SQLRETURN       rc      = SQL_SUCCESS;
  struct sqlca    sqlca;
  struct sqlca    *pSQLCA = &sqlca;
  int  code ;
  char state [6];
  char errp  [9];
  char tok   [40];
  int  count, len, start, end, i;
```

Figure 22. An application that retrieves diagnostic information (Part 4 of 5)

```
  if ((rc = SQLGetSQLCA (hEnv ,
                         hDbc ,
                         hStmt,
                         pSQLCA)) != SQL_SUCCESS)
  {
    (void) printf ("**** SQLGetSQLCA failed Return Code = %d.\n", rc);
    goto exit;
  }

  code = (int) pSQLCA->sqlcode;
  memcpy (state, pSQLCA->sqlstate, 5);
  state [5] = '\0';

  (void) printf ("**** sqlcode = %d, sqlstate = %s.\n", code, state);

  memcpy (errp, pSQLCA->sqlerrp, 8);
  errp [8] = '\0';
  (void) printf ("**** sqlerrp = %s.\n", errp);

  if (pSQLCA->sqlerrml == 0)
    (void) printf ("**** No tokens.\n");
  else
  {
    for (len = 0, count = 0; len < pSQLCA->sqlerrml; len = ++end)
    {
      start = end = len;
      while ((pSQLCA->sqlerrmc [end] != 0XFF) &&;
             (end < pSQLCA->sqlerrml))

        end++;
      if (start != end)
      {
        memcpy (tok, &pSQLCA->sqlerrmc[start],
                                                      (end-start));
        tok [end-start+1] = '\0';
        (void) printf ("**** Token # %d = %s.\n", count++, tok);
      }
    }
  }

  for (i = 0; i <= 5; i++)
    (void) printf ("**** sqlerrd # %d = %d.\n", i+1, pSQLCA->sqlerrd_i]);

  for (i = 0; i <= 10; i++)
    (void) printf ("**** sqwarn # %d = %c.\n", i+1, pSQLCA->sqlwarn_i]);

  exit:
  return;
}
```

Figure 22. An application that retrieves diagnostic information (Part 5 of 5)

Related functions

The following functions relate to SQLGetSQLCA() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetSQLCA() in your applications.

- “SQLGetDiagRec() - Get multiple field settings of diagnostic record” on page 221

SQLGetStmtAttr() - Get current setting of a statement attribute

Purpose

Table 130. SQLGetStmtAttr() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLGetStmtAttr() returns the current setting of a statement attribute. To set statement attributes, use SQLSetStmtAttr().

Syntax

```
SQLRETURN  SQLGetStmtAttr (SQLHSTMT         StatementHandle,
                           SQLINTEGER       Attribute,
                           SQLPOINTER       ValuePtr,
                           SQLINTEGER       BufferLength,
                           SQLINTEGER       *StringLengthPtr);
```

Function arguments

Table 131 lists the data type, use, and description for each argument in this function.

Table 131. SQLGetStmtAttr() arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Specifies a connection handle.
SQLINTEGER	*Attribute*	input	Specifies the statement attribute to retrieve. Refer to Table 205 on page 368 for a complete list of these attributes.
SQLPOINTER	*ValuePtr*	output	Points to a buffer in which to return the current value of the attribute specified by the *Attribute* argument. The value that is returned into this buffer is a 32-bit unsigned integer value or a nul-terminated character string. If the a driver-specific value is specified for the *Attribute* argument, a signed integer might be returned.
SQLINTEGER	*BufferLength*	input	The value that you specify for this argument depends which of the following types of attributes you query: • For ODBC-defined attributes: – If the *ValuePtr* argument points to a character string, the *BufferLength* argument specifies the length (in bytes) of the buffer to which the *ValuePtr* argument points. – If the *ValuePtr* argument points to an integer, the *BufferLength* argument is ignored. • For driver-defined attributes (IBM extension): – If the *ValuePtr* argument points to a character string, the *BufferLength* argument specifies the length (in bytes) of the buffer to which the *ValuePtr* argument points, or specifies SQL_NTS for nul-terminated strings. If SQL_NTS is specified, the driver assumes the length of buffer to which the *ValuePtr* argument points to be SQL_MAX_OPTIONS_STRING_LENGTH bytes (which excludes the nul-terminator). – If the *ValuePtr* argument points to an integer, the *BufferLength* argument is ignored.

Table 131. SQLGetStmtAttr() arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER *	*StringLengthPtr*	output	Points to a buffer in which to return the total number of bytes (excluding the number of bytes returned for the nul-termination character) available to return in the buffer to which the *ValuePtr* argument points. • If the *ValuePtr* argument specifies a null pointer, no length is returned. • If the value 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 nul-termination character and is nul-terminated by DB2 ODBC. • If the *Attribute* argument does not denote a string, DB2 ODBC ignores the *BufferLength* argument and does not return a value in the buffer to which the *StringLengthPtr* argument points.

Usage

SQLGetStmtAttr() returns the current setting of a statement attribute. You set these attributes using the SQLSetStmtAttr() function. For a list of valid statement attributes, refer to Table 205 on page 368.

For information about overriding DB2 CCSIDs from DSNHDECP, see “Usage” on page 367.

Return codes

After you call SQLGetStmtAttr(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 132 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 132. SQLGetStmtAttr() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (SQLGetStmtAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
01004	Data truncated.	The data that is returned in the buffer to which the *ValuePtr* argument points is truncated to be the length (in bytes) of the value that the *BufferLength* argument specifies, minus the length of a nul-terminator. The length (in bytes) of the untruncated string value is returned in the buffer to which the *StringLengthPtr* argument points. (SQLGetStmtAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)

Table 132. SQLGetStmtAttr() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY000	General error.	An error occurred for which no specific SQLSTATE exists. The error message that SQLGetDiagRec() returns describes the specific error and the cause of that error.
HY001	Memory allocation failure.	DB2 ODBC can not allocate memory that is required to support execution or completion of the function.
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() is called on the statement handle and returns SQL_NEED_DATA. This function is called before data is sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory that is required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the *BufferLength* argument is less than 0.
HY092	Option type out of range.	The value specified for the *Attribute* argument is not valid for this version of DB2 ODBC.
HYC00	Driver not capable.	The value specified for the *Attribute* argument is a valid connection or statement attribute for the version of the DB2 ODBC driver, but is not supported by the data source.

Restrictions

None.

Example

The following example uses SQLGetStmtAttr() to retrieve the current value of a statement attribute:

```
SQLINTEGER cursor_hold;
rc = SQLGetStmtAttr( hstmt, SQL_ATTR_CURSOR_HOLD,
                                                        &cursor_hold, 0, NULL ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
printf( "\nCursor With Hold is: " ) ;
if ( cursor_hold == SQL_CURSOR_HOLD_ON )
  printf( "ON\n" ) ;
else
  printf( "OFF\n" ) ;
```

Related functions

The following functions relate to SQLGetStmtAttr() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetStmtAttr() in your applications.

- “SQLGetConnectAttr() - Get current attribute setting” on page 196
- “SQLSetConnectAttr() - Set connection attributes” on page 346
- “SQLSetStmtAttr() - Set statement attributes” on page 367

SQLGetStmtOption() - Return current setting of a statement option

Purpose

Table 133. SQLGetStmtOption() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	No

In the current version of DB2 ODBC, SQLGetStmtAttr() replaces SQLGetStmtOption(). See “SQLGetStmtAttr() - Get current setting of a statement attribute” on page 272 for more information.

Although DB2 ODBC supports SQLGetStmtOption() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLGetStmtOption() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLGetStmtOption (SQLHSTMT         hstmt,
                              SQLUSMALLINT     fOption,
                              SQLPOINTER       pvParam);
```

Function arguments

Table 134 lists the data type, use, and description for each argument in this function.

Table 134. SQLGetStmtOption() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle.
SQLUSMALLINT	*fOption*	input	Specifies the attribute to set.
SQLPOINTER	*pvParam*	output	Specifies the value of the attribute. Depending on the value of *fOption* this can be a 32-bit integer value, or a pointer to a nul-terminated character string. The maximum length of any character string returned is SQL_MAX_OPTION_STRING_LENGTH bytes (which excludes the nul-terminator).

SQLGetSubString() - Retrieve portion of a string value

Purpose

Table 135. SQLGetSubString() specifications

ODBC	X/OPEN CLI	ISO CLI
No	No	No

SQLGetSubString() retrieves a portion of a large object value, referenced by a LOB locator that the server returns (returned by a fetch or a previous SQLGetSubString() call) during the current transaction.

Syntax

```
SQLRETURN  SQLGetSubString   (SQLHSTMT          hstmt,
                              SQLSMALLINT       LocatorCType,
                              SQLINTEGER        SourceLocator,
                              SQLUINTEGER       FromPosition,
                              SQLUINTEGER       ForLength,
                              SQLSMALLINT       TargetCType,
                              SQLPOINTER        rgbValue,
                              SQLINTEGER        cbValueMax,
                              SQLINTEGER  FAR   *StringLength,
                              SQLINTEGER  FAR   *IndicatorValue);
```

Function arguments

Table 136 lists the data type, use, and description for each argument in this function.

Table 136. SQLGetSubString() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle. This can be any statement handle that is allocated but does not currently have a prepared statement assigned to it.
SQLSMALLINT	*LocatorCType*	input	Specifies the C type of the source LOB locator with one of the following values: • SQL_C_BLOB_LOCATOR for BLOB data • SQL_C_CLOB_LOCATOR for CLOB data • SQL_C_DBCLOB_LOCATOR for DBCLOB data
SQLINTEGER	*Locator*	input	Specifies the source LOB locator value.
SQLUINTEGER	*FromPosition*	input	Specifies the position at which the string that is retrieved begins. For BLOBs and CLOBs, this is the position of the first byte the function returns. For DBCLOBs, this is the first character. The start-byte or start-character is numbered 1.
SQLUINTEGER	*ForLength*	input	Specifies the length of the string that SQLGetSubString() retrieves. For BLOBs and CLOBs, this is the length in bytes. For DBCLOBs, this is the length in characters. If the value that the *FromPosition* argument specifies is less than the length of the source string, but *FromPosition* + *ForLength* -1 extends beyond the position of 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).

Table 136. SQLGetSubString() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*TargetCType*	input	Specifies the target C data type for the string that is retrieved into the buffer to which the *rgbValue* argument points. This target can be a LOB locator C buffer of one of the following types: • SQL_C_CLOB_LOCATOR • SQL_C_BLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR Or, the target can be a C string variable of one of the following types: • SQL_C_CHAR for CLOB data • SQL_C_BINARY for BLOB data • SQL_C_DBCHAR for DBCLOB data
SQLPOINTER	*rgbValue*	output	Pointer to the buffer where the retrieved string value or a LOB locator is stored.
SQLINTEGER	*cbValueMax*	input	Specifies the maximum size (in bytes) of the buffer to which the *rgbValue* argument points.
SQLINTEGER *	*StringLength*	output	If the target C buffer type is intended for a binary or character string variable, not a locator value, this argument points to the length (in bytes[1]) of the substring that is retrieved. If a null pointer is specified, no value is returned.
SQLINTEGER *	*IndicatorValue*	output	Always returns zero.

Note:

1. This is in bytes even for DBCLOB data.

Usage

Use SQLGetSubString() to obtain any portion of the string that a LOB locator represents. The target for this substring can be one of the following objects:

- An appropriate C string variable.
- A new LOB value that is created on the server. The LOB locator for this value can be assigned to a target application variable on the client.

You can use SQLGetSubString() as an alternative to SQLGetData() for retrieving data in pieces. To use SQLGetSubString() to retrieve data in pieces, you first bind a column to a LOB locator. You then use this LOB locator to fetch the LOB value as a whole or in pieces.

The *Locator* argument can contain any valid LOB locator that you do not explicitly free using the FREE LOCATOR statement or is not implicitly freed because the transaction during which it was created has completed.

The statement handle must not be associated with any prepared statements or catalog function calls.

Return codes

After you call SQLGetSubString(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 137 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 137. SQLGetSubString() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The amount of returned data is longer than *cbValueMax*. Actual length, in bytes, that is available for return is stored in *StringLength*.
07006	Invalid conversion.	This SQLSTATE is returned for one or more of the following reasons: • The value specified for *TargetCType* is 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).
0F001	The LOB token variable does not currently represent any value.	The value specified for *Locator* or *SearchLocator* is not currently a LOB locator.
22011	A substring error occurred.	*FromPosition* is greater than the length of the source string.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY003	Program type out of range.	*LocatorCType* is not one of the following: • SQL_C_CLOB_LOCATOR • SQL_C_BLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY024	Invalid argument value.	The value specified for *FromPosition* or for *ForLength* is not a positive integer.
HY090	Invalid string or buffer length.	The value of *cbValueMax* is 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 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

See Figure 21 on page 264.

Related functions

The following functions relate to SQLGetSubString() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetSubString() in your applications.

- “SQLBindCol() - Bind a column to an application variable” on page 78
- “SQLExtendedFetch() - Fetch an array of rows” on page 163
- “SQLFetch() - Fetch the next row” on page 171
- “SQLGetLength() - Retrieve length of a string value” on page 258
- “SQLGetSubString() - Retrieve portion of a string value” on page 276

SQLGetTypeInfo() - Get data type information

Purpose

Table 138. SQLGetTypeInfo() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLGetTypeInfo() returns information about the data types that are supported by the DBMSs that are associated with DB2 ODBC. This information is returned in an SQL result set. The columns of this result set can be received using the same functions that you use to process a query.

Syntax

```
SQLRETURN   SQLGetTypeInfo   (SQLHSTMT         hstmt,
                              SQLSMALLINT      fSqlType);
```

Function arguments

Table 139 lists the data type, use, and description for each argument in this function.

Table 139. SQLGetTypeInfo() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle.

Table 139. SQLGetTypeInfo() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*fSqlType*	input	Specifies the SQL data type that is queried. The following values that specify data types are supported: • SQL_ALL_TYPES • SQL_BINARY • SQL_BLOB • SQL_CHAR • SQL_CLOB • SQL_DBCLOB • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_ROWID • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC If the value SQL_ALL_TYPES is specified, information about all supported data types is returned in ascending order by TYPE_NAME. All unsupported data types are absent from the result set.

Usage

Because SQLGetTypeInfo() generates a result set it is essentially equivalent to executing a query. Like a query, calling SQLGetTypeInfo() generates a cursor and begins a transaction. To prepare and execute another statement on this statement handle, the cursor must be closed.

If you call SQLGetTypeInfo() with an invalid value in the *fSqlType* argument, an empty result set is returned.

Table 140 on page 282 describes each column in the result set that this function generates.

Although new columns might be added and the names of the existing columns might be changed in future releases, the position of the current columns does not change. The data types that are returned are those that can be used in a CREATE TABLE or ALTER TABLE, statement. Nonpersistent data types such as the locator data types are not part of the returned result set. User-defined data types are not returned either.

Table 140. Columns returned by SQLGetTypeInfo()

Position	Column name	Data type	Description
1	TYPE_NAME	VARCHAR(128) NOT NULL	Contains a character representation of the SQL DDL data type name. For example, VARCHAR, BLOB, DATE, INTEGER.
2	DATA_TYPE	SMALLINT NOT NULL	Contains the SQL data type definition values. For example, SQL_VARCHAR, SQL_BLOB, SQL_TYPE_DATE, SQL_INTEGER.
3	COLUMN_SIZE	INTEGER	If the data type is a character or binary string, then this column contains the maximum length in bytes. If this data type is a graphic (DBCS) string, this column contains 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 characters. For numeric data types, this column contains the total number of digits.
4	LITERAL_PREFIX	VARCHAR(128)	Contains the 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.
5	LITERAL_SUFFIX	VARCHAR(128)	Contains the 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.
6	CREATE_PARAMS	VARCHAR(128)	Contains a list of values, that are separated by commas. These values correspond to each parameter that you can specify for a data type in a CREATE TABLE or an ALTER TABLE SQL statement. One or more of the following values appear in this result-set column: • LENGTH, which indicates you can specify a length for the data type in the TYPE_NAME column • PRECISION, which indicates you can specify the precision for the data type in the TYPE_NAME column • SCALE, which indicates you can specify a scale for the data type in the TYPE_NAME column • A null indicator, which indicates you cannot specify any parameters for the data type in the TYPE_NAME column **Tip:** The CREATE_PARAMS column enables you to customize the interface of DDL builders in your applications. A *DDL builder* is a piece of your application that creates database objects, such as tables. Use the CREATE_PARAMS to determine the number of arguments that are required to define a data type, then use localized text to label the controls on the DDL builder.

Table 140. Columns returned by SQLGetTypeInfo() (continued)

Position	Column name	Data type	Description
7	NULLABLE	SMALLINT NOT NULL	Indicates whether the data type accepts a null value. This column contains one of the following values: • SQL_NO_NULLS, which indicates that null values are disallowed • SQL_NULLABLE, which indicates that null values are allowed
8	CASE_SENSITIVE	SMALLINT NOT NULL	Indicates whether the data type can be treated as case sensitive for collation purposes. This column contains one of the following values: • SQL_TRUE, which indicates case sensitivity • SQL_FALSE, which indicates no case sensitivity
9	SEARCHABLE	SMALLINT NOT NULL	Indicates how the data type is used in a WHERE clause. This column contains one of the following values: • SQL_UNSEARCHABLE, which indicates that you cannot use the data type in a WHERE clause • SQL_LIKE_ONLY, which indicates that you can use the data type in a WHERE clause, but only with the LIKE predicate. • SQL_ALL_EXCEPT_LIKE, which indicates that you can use the data type in a WHERE clause with all comparison operators except LIKE. • SQL_SEARCHABLE, which indicates that you can use the data type in a WHERE clause with any comparison operator.
10	UNSIGNED_ATTRIBUTE	SMALLINT	Indicates whether the data type is unsigned. This column contains one of the following values: • SQL_TRUE, which indicates that the data type is unsigned • SQL_FALSE, which indicates the data type is signed • NULL, which indicates this attribute does not apply to the data type
11	FIXED_PREC_SCALE	SMALLINT NOT NULL	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.
12	AUTO_INCREMENT	SMALLINT	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.
13	LOCAL_TYPE_NAME	VARCHAR(128)	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 contains a null indicator. 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.
14	MINIMUM_SCALE	SMALLINT	Contains 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.

Table 140. Columns returned by SQLGetTypeInfo() (continued)

Position	Column name	Data type	Description
15	MAXIMUM_SCALE	SMALLINT	Contains 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.

Return codes

After you call SQLGetTypeInfo(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 141 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 141. SQLGetTypeInfo() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is open on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY004	Invalid SQL data type.	An invalid value for the *fSqlType* argument is specified.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)

Restrictions

The following ODBC specified SQL data types (and their corresponding *fSqlType* define values) are not supported by any IBM RDBMS:

Data type	***fSqlType***
TINY INT	SQL_TINYINT
BIG INT	SQL_BIGINT
BIT	SQL_BIT

Example

Figure 23 on page 285 shows an application that uses SQLGetTypeInfo() to check which ODBC data types the DBMS supports.

```
 /******************************************************************/
 /*  Invoke SQLGetTypeInfo to retrieve SQL data types supported.   */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

  /******************************************************************/
  /* Invoke SQLGetTypeInfo to retrieve all SQL data types supported */
  /* by data source.                                                */
  /******************************************************************/

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;

   (void) printf ("**** Entering CLIP06.\n\n");

  /*****************************************************************/
  /* Allocate environment handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate connection handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;

  /*****************************************************************/
  /* CONNECT TO data source (STLEC1)                               */
  /*****************************************************************/

   RETCODE = SQLConnect(hDbc,        // Connect handle
                     (SQLCHAR *) "STLEC1", // DSN
                     SQL_NTS,     // DSN is nul-terminated
                     NULL,        // Null UID
                     0   ,
                     NULL,        // Null Auth string
                     0);

   if( RETCODE != SQL_SUCCESS )      // Connect failed
     goto dberror;
```

Figure 23. An application that checks data types the current server supports (Part 1 of 4)

```
/******************************************************************/
/* Retrieve SQL data types from DSN                               */
/******************************************************************/
// local variables to Bind to retrieve TYPE_NAME, DATA_TYPE,
// COLUMN_SIZE and NULLABLE

 struct                    // TYPE_NAME is VARCHAR(128)
 {
   SQLSMALLINT  length;
   SQLCHAR      name [128];
   SQLINTEGER   ind;
 } typename;

 SQLSMALLINT data_type;    // DATA_TYPE is SMALLINT
 SQLINTEGER  data_type_ind;
 SQLINTEGER  column_size;  // COLUMN_SIZE is integer
 SQLINTEGER  column_size_ind;
 SQLSMALLINT nullable;     // NULLABLE is SMALLINT
 SQLINTEGER  nullable_ind;

/******************************************************************/
/* Allocate statement handle                                      */
/******************************************************************/

 rc = SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt);

 if (rc != SQL_SUCCESS)
   goto exit;

/******************************************************************/
/*                                                                */
/* Retrieve native SQL types from DSN ------------>               */
/*                                                                */
/*  The result set consists of 15 columns. We only bind           */
/*  TYPE_NAME, DATA_TYPE, COLUMN_SIZE and NULLABLE. Note: Need    */
/*  not bind all columns of result set -- only those required.    */
/*                                                                */
/******************************************************************/

 rc = SQLGetTypeInfo (hStmt,
                     SQL_ALL_TYPES);

 if (rc != SQL_SUCCESS)
   goto exit;

 rc = SQLBindCol (hStmt,           // bind TYPE_NAME
                 1,
                 SQL_CHAR,
                 (SQLPOINTER) typename.name,
                 128,
                 &typename.ind);

 if (rc != SQL_SUCCESS)
   goto exit;

 rc = SQLBindCol (hStmt,           // bind DATA_NAME
                 2,
                 SQL_C_DEFAULT,
                 (SQLPOINTER) &data_type,
                 sizeof(data_type),
                 &data_type_ind);

 if (rc != SQL_SUCCESS)
   goto exit;
```

Figure 23. An application that checks data types the current server supports (Part 2 of 4)

```
 rc = SQLBindCol (hStmt,           // bind COLUMN_SIZE
                  3,
                  SQL_C_DEFAULT,
                  (SQLPOINTER) &column_size,
                  sizeof(column_size),
                  &column_size_ind);

 if (rc != SQL_SUCCESS)
   goto exit;

 rc = SQLBindCol (hStmt,           // bind NULLABLE
                  7,
                  SQL_C_DEFAULT,
                  (SQLPOINTER) &nullable,
                  sizeof(nullable),
                  &nullable_ind);

 if (rc != SQL_SUCCESS)
   goto exit;


/*****************************************************************/
/* Fetch all native DSN SQL Types and print Type Name, Type,     */
/* Precision and nullability.                                    */
/*****************************************************************/

 while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
 {
   (void) printf ("**** Type Name is %s. Type is %d. Precision is %d.",
                  typename.name,
                  data_type,
                  column_size);
   if (nullable == SQL_NULLABLE)
     (void) printf (" Type is nullable.\n");
   else
     (void) printf (" Type is not nullable.\n");
 }

 if (rc == SQL_NO_DATA_FOUND)   // if result set exhausted reset
   rc = SQL_SUCCESS;            // rc to OK

/*****************************************************************/
/* Free statement handle                                         */
/*****************************************************************/

 rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);


 if (RETCODE != SQL_SUCCESS)       // An advertised API failed
   goto dberror;

/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/

 RETCODE = SQLDisconnect(hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
```

Figure 23. An application that checks data types the current server supports (Part 3 of 4)

```
  /****************************************************************/
  /* Deallocate connection handle                                 */
  /****************************************************************/

   RETCODE = SQLFreeHandle(SQL_HANDLE_DBC, hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /****************************************************************/
  /* Free environment handle                                      */
  /****************************************************************/

   RETCODE = SQLFreeHandle(SQL_HANDLE_ENV, hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;
     dberror:
     RETCODE=12;

     exit:

     (void) printf ("**** Exiting  CLIP06.\n\n");

     return(RETCODE);
}
```

Figure 23. An application that checks data types the current server supports (Part 4 of 4)

Related functions

The following functions relate to SQLGetTypeInfo() calls. Refer to the descriptions of these functions for more information about how you can use SQLGetTypeInfo() in your applications.

- “SQLColAttribute() - Get column attributes” on page 101
- “SQLExtendedFetch() - Fetch an array of rows” on page 163
- “SQLGetInfo() - Get general information” on page 234

SQLMoreResults() - Check for more result sets

Purpose

Table 142. SQLMoreResults() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

SQLMoreResults() determines whether there is more information available on a statement handle which has been associated with one of the following actions:
- Array input of parameter values for a query
- A stored procedure that is returning result sets

Syntax

```
SQLRETURN   SQLMoreResults   (SQLHSTMT         hstmt);
```

Function arguments

Table 143 lists the data type, use, and description for each argument in this function.

Table 143. SQLMoreResults() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle on which results are returned.

Usage

Use this function to return a sequence of result sets after you execute of one of the following actions:
- A parameterized query with an array of input parameter values that SQLParamOptions() and SQLBindParameter() specify
- A stored procedure that contains SQL queries that leaves open cursors on the result sets that it generates (result sets are accessible when a stored procedure has finished execution if cursors on these result sets remain open)

See “Using arrays to pass parameter values” on page 414 and “Returning result sets from stored procedures” on page 431 for more information.

After you completely process a result set, call SQLMoreResults() to determine if another result set is available. When you call SQLMoreResults(), this function discards rows that were not fetched in the current result set by closing the cursor. If another result set is available SQLMoreResults() returns SQL_SUCCESS.

If all the result sets have been processed, SQLMoreResults() returns SQL_NO_DATA_FOUND.

If you call SQLFreeStmt() with the *fOption* argument set to SQL_CLOSE or you call SQLFreeHandle() is called with the *HandleType* argument set to SQL_HANDLE_STMT, these functions discard all pending result sets for the statement handle on which they are called.

Return codes

After you call SQLMoreResults(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 144 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 144. SQLMoreResults() SQLSTATEs

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.

Additionally, SQLMoreResults() can return all SQLSTATEs that are associated with SQLExecDirect() except for **HY**009, **HY**014, and **HY**090. See Table 73 on page 156. for these additional SQLSTATEs.

Restrictions

The ODBC specification of SQLMoreResults() allows row-counts associated with the execution of parameterized INSERT, UPDATE, and DELETE statements with arrays of input parameter values to be returned. However, DB2 ODBC does not support the return of this count information.

Example

Figure 24 on page 291 shows an application that uses SQLMoreResults() to check for additional result sets.

```
/* ... */
#define NUM_CUSTOMERS 25
    SQLCHAR         stmt[] =
    {  "WITH "  /* Common Table expression (or Define Inline View) */
         "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 cust_num = CNUM(cast (? 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
    };

#define  NUM_CUSTOMERS sizeof(Cust)/sizeof(SQLINTEGER)

    /* Row-wise (Includes buffer for both column data and length) */
    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[ROWSET_SIZE];

    SQLUINTEGER     pirow = 0;
    SQLUINTEGER     pcrow;
    SQLINTEGER      i;
    SQLINTEGER      j;
/* ... */
```

Figure 24. An application that checks for additional result sets (Part 1 of 2)

```
    /* Get details and total for each order row-wise */
    rc = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

    rc = SQLParamOptions(hstmt, NUM_CUSTOMERS, &pirow);

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

    rc = SQLExecDirect(hstmt, stmt, SQL_NTS);

    /* SQL_ROWSET_SIZE sets the max number of result rows to fetch each time */
    rc = SQLSetStmtAttr(hstmt, SQL_ATTR_ROWSET_SIZE, ROWSET_SIZE, 0);

    /* Set size of one row, used for row-wise binding only */
    rc = SQLSetStmtAttr(hstmt, SQL_ATTR_BIND_TYPE, (void*)sizeof(Ord)/ROW_SIZE, 0);

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

    /* Bind remaining columns ... */
/* ... */
    /* NOTE: This sample assumes that an order never has more
             rows than ROWSET_SIZE.  A check should be added below to call
             SQLExtendedFetch multiple times for each result set.
    */
    do  /* for each result set .... */
    { rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, NULL);

      if (pcrow > 0) /* if 1 or more rows in the result set */
      {
        i = j = 0;

        printf("**************************************\n");
        printf("Orders for Customer: %ld\n", Ord[0].Cust_Num);
        printf("**************************************\n");

        while (i < pcrow)
        {   printf("\nOrder #: %ld\n", Ord[i].Ord_Num);
            printf("     Product  Quantity         Price\n");
            printf("     -------- ---------------- ------------\n");
            j = i;
            while (Ord[j].Ord_Num == Ord[i].Ord_Num)
            {   printf("    %8ld %16.7lf %12.2lf\n",
                       Ord[i].Prod_Num, Ord[i].Quant, Ord[i].Amount);
                i++;
            }
            printf("                               ============\n");
            printf("                              %12.2lf\n", Ord[j].Total);
        } /* end while */
      } /* end if */
    }
    while ( SQLMoreResults(hstmt) == SQL_SUCCESS);
/* ... */
```

Figure 24. An application that checks for additional result sets (Part 2 of 2)

Related functions

The following functions relate to SQLMoreResults() calls. Refer to the descriptions of these functions for more information about how you can use SQLMoreResults() in your applications.

- “SQLCloseCursor() - Close a cursor and discard pending results” on page 99

- “SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator” on page 85
- “SQLParamOptions() - Specify an input array for a parameter” on page 304

SQLNativeSql() - Get native SQL text

Purpose

Table 145. SQLNativeSql() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

SQLNativeSql() is used to show how DB2 ODBC interprets vendor escape clauses. If the original SQL string passed in by the application contains vendor escape clause sequences, DB2 ODBC passes a transformed SQL string to the data source (with vendor escape clauses either converted or discarded, as appropriate).

Syntax

```
SQLRETURN   SQLNativeSql     (SQLHDBC          hdbc,
                               SQLCHAR      FAR  *szSqlStrIn,
                               SQLINTEGER        cbSqlStrIn,
                               SQLCHAR      FAR  *szSqlStr,
                               SQLINTEGER        cbSqlStrMax,
                               SQLINTEGER   FAR  *pcbSqlStr);
```

Function arguments

Table 146 lists the data type, use, and description for each argument in this function.

Table 146. SQLNativeSql() arguments

Data type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Specifies the connection handle.
SQLCHAR *	*szSqlStrIn*	input	Points to a buffer that contains the input SQL string.
SQLINTEGER	*cbSqlStrIn*	input	Specifies the length, in bytes, of the buffer to which the *szSqlStrIn* argument points.
SQLCHAR *	*szSqlStr*	output	Points to buffer that returns the transformed output string.
SQLINTEGER	*cbSqlStrMax*	input	Specifies the size of the buffer to which the *szSqlStr* argument points.
SQLINTEGER *	*pcbSqlStr*	output	Points to a buffer that returns the total number of bytes (excluding the nul-terminator) that the complete output string requires. If this string requires a number of bytes that is greater than or equal to the value in the *cbSqlStrMax* argument, the output string is truncated to *cbSqlStrMax - 1* bytes.

Usage

Call this function when you want to examine or display a transformed SQL string that is passed to the data source by DB2 ODBC. Translation (mapping) only occurs if the input SQL statement string contains vendor escape clause sequences. For more information on vendor escape clause sequences, see “Using vendor escape clauses” on page 465

DB2 ODBC can only detect vendor escape clause syntax errors; because DB2 ODBC does not pass the transformed SQL string to the data source for preparation, syntax errors that are detected by the DBMS are not generated for the input SQL string at this time. (The statement is not passed to the data source for preparation because the preparation can potentially cause the initiation of a transaction.)

Return codes

After you call SQLNativeSql(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 147 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 147. SQLNativeSql() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The output string is truncated because the buffer to which the *szSqlStr* argument points is not large enough to contain the entire SQL string. The argument *pcbSqlStr* contains the total length, in bytes, of the untruncated SQL string. (SQLNativeSql() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
08003	Connection is closed.	The *hdbc* argument does not reference an open database connection.
37000	Invalid SQL syntax.	The input SQL string that the *szSqlStrIn* argument specifies contains a syntax error in the escape sequence.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	This SQLSTATE is returned for one or more of the following reasons: • The argument *szSqlStrIn* is a null pointer. • The argument *szSqlStr* is a null pointer.
HY090	Invalid string or buffer length.	This SQLSTATE is returned for one or more of the following reasons: • The argument *cbSqlStrIn* specifies a value that is less than 0 and not equal to SQL_NTS. • The argument *cbSqlStrMax* specifies a value that is less than 0.

Restrictions

None.

Example

Figure 25 on page 296 shows an application that uses SQLNativeSql() to print the final version of an SQL statement that contains vendor escape clauses.

```
/* ... */
    SQLCHAR         in_stmt[1024];
    SQLCHAR         out_stmt[1024];
    SQLSMALLINT     pcPar;
    SQLINTEGER      indicator;
/* ... */
    /* Prompt for a statement to prepare */
    printf("Enter an SQL statement: \n");
    gets(in_stmt);

    /* prepare the statement */
    rc = SQLPrepare(hstmt, in_stmt, SQL_NTS);

    SQLNumParams(hstmt, &pcPar);

    SQLNativeSql(hstmt, in_stmt, SQL_NTS, out_stmt, 1024, &indicator);

    if (indicator == SQL_NULL_DATA)
    {  printf("Invalid statement\n"); }
    else
    {  printf(" Input Statement: \n %s \n", in_stmt);
       printf("Output Statement: \n %s \n", out_stmt);
       printf("Number of Parameter Markers = %ld\n", pcPar);
    }
    rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
/* ... */
```

Figure 25. An application that prints a translated vendor escape clause

Related functions

No functions directly relate to SQLNativeSql().

SQLNumParams() - Get number of parameters in a SQL statement

Purpose

Table 148. SQLNumParams() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

SQLNumParams() returns the number of parameter markers that are in a SQL statement.

Syntax

```
SQLRETURN   SQLNumParams     (SQLHSTMT         hstmt,
                              SQLSMALLINT  FAR  *pcpar);
```

Function arguments

Table 149 lists the data type, use, and description for each argument in this function.

Table 149. SQLNumParams() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle.
SQLSMALLINT *	*pcpar*	output	Points to a buffer that returns the number of parameters in the statement.

Usage

You call this function to determine how many SQLBindParameter() calls are necessary for the SQL statement that is associated with a statement handle.

You can call this function only after you prepare the statement associated with the *hstmt* argument. If the statement does not contain any parameter markers, the buffer to which the *pcpar* argument points is set to 0.

Return codes

After you call SQLNumParams(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 150 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 150. SQLNumParams() SQLSTATEs

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.

Table 150. SQLNumParams() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	The *pcpar* argument specifies a null pointer.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • SQLNumParams() is called before SQLPrepare() for the statement to which the *hstmt* argument refers. • SQLNumParams() is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.

Restrictions

None.

Example

See Figure 25 on page 296.

Related functions

The following functions relate to SQLNumParams() calls. Refer to the descriptions of these functions for more information about how you can use SQLNumParams() in your applications.

- "SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator" on page 85
- "SQLPrepare() - Prepare a statement" on page 306

SQLNumResultCols() - Get number of result columns

Purpose

Table 151. SQLNumResultCols() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLNumResultCols() returns the number of columns in the result set that is 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          hstmt,
                              SQLSMALLINT FAR   *pccol);
```

Function arguments

Table 152 lists the data type, use, and description for each argument in this function.

Table 152. SQLNumResultCols() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle.
SQLSMALLINT *	*pccol*	output	Points to a buffer that returns the number of columns in the result set.

Usage

You call this function to determine how many SQLBindCol() or SQLGetData() calls are necessary for the SQL result set that is associated with a statement handle.

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

After you call SQLNumResultCols(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 153 on page 300 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 153. SQLNumResultCols() SQLSTATEs

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	*pcccol* is a null pointer.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • The function is called prior to calling SQLPrepare() or SQLExecDirect() for the *hstmt*. • The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.

Restrictions

None.

Example

See Figure 13 on page 134.

Related functions

The following functions relate to SQLNumResultCols() calls. Refer to the descriptions of these functions for more information about how you can use SQLNumResultCols() in your applications.

- "SQLBindCol() - Bind a column to an application variable" on page 78
- "SQLColAttribute() - Get column attributes" on page 101
- "SQLDescribeCol() - Describe column attributes" on page 131
- "SQLExecDirect() - Execute a statement directly" on page 154
- "SQLGetData() - Get data from a column" on page 207
- "SQLPrepare() - Prepare a statement" on page 306

SQLParamData() - Get next parameter for which a data value is needed

Purpose

Table 154. SQLParamData() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLParamData() is used in conjunction with SQLPutData() to send long data in pieces. You can also use this function to send fixed-length data. For a description of the exact sequence of this input method, see “Sending or retrieving long data values in pieces” on page 412.

Syntax

```
SQLRETURN   SQLParamData     (SQLHSTMT         hstmt,
                              SQLPOINTER  FAR   *prgbValue);
```

Function arguments

Table 155 lists the data type, use, and description for each argument in this function.

Table 155. SQLParamData() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies the statement handle.
SQLPOINTER *	*prgbValue*	output	Points to the buffer that the *rgbValue* argument in the SQLBindParameter() call indicates.

Usage

SQLParamData() returns SQL_NEED_DATA if there is at least one SQL_DATA_AT_EXEC parameter for which data is not assigned. This function returns an application provided value in *prgbValue*, which a previous SQLBindParameter() call supplies. When you send long data in pieces, you call SQLPutData() one or more times. After the SQLPutData() calls, you call SQLParamData() to signal all data for the current parameter is sent and to advance to the next SQL_DATA_AT_EXEC parameter.

SQLParamData() returns SQL_SUCCESS 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, SQLParamData() returns SQL_ERROR.

SQLParamData() returns SQL_NEED_DATA when you advance to the next SQL_DATA_AT_EXEC parameter. You can call only SQLPutData() or SQLCancel() at this point in the transaction; all other function calls that use the same statement handle that the *hstmt* argument specifies will fail. Additionally, all functions that reference the parent connection handle of the statement that the *hstmt* argument references fail if they change any attribute or state of that connection. Because functions that reference the parent connection handle fail, do not use the following functions on the parent connection handle during an SQL_NEED_DATA sequence:

- SQLAllocHandle()
- SQLSetConnectAttr()

- SQLNativeSql()
- SQLEndTran()

These functions return SQL_ERROR with SQLSTATE **HY**010 and the processing of the SQL_DATA_AT_EXEC parameters is not affected if these functions are called during an SQL_NEED_DATA sequence.

Return codes

After you call SQLParamData(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

SQLParamData() can return any SQLSTATE that SQLExecDirect() and SQLExecute() generate. Table 156 lists the additional SQLSTATEs SQLParamData() can generate with a description and explanation for each value.

Table 156. SQLParamData() SQLSTATEs

SQLSTATE	Description	Explanation
40001	Transaction rollback.	The transaction to which this SQL statement belongs is rolled back due to a deadlock or timeout.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • SQLParamData() is called out of sequence. This call is only valid after an SQLExecDirect() or an SQLExecute(), or after an SQLPutData() call. • SQLParamData() is called after an SQLExecDirect() or an SQLExecute() call, but no SQL_DATA_AT_EXEC parameters require processing.

Restrictions

None.

Example

See Figure 19 on page 213.

Related functions

The following functions relate to SQLParamData() calls. Refer to the descriptions of these functions for more information about how you can use SQLParamData() in your applications.
- "SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator" on page 85
- "SQLCancel() - Cancel statement" on page 97

- “SQLExecDirect() - Execute a statement directly” on page 154
- “SQLPutData() - Pass a data value for a parameter” on page 335
- “SQLSetParam() - Bind a parameter marker to a buffer” on page 364

SQLParamOptions() - Specify an input array for a parameter

Purpose

Table 157. SQLParamOptions() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	No	No

SQLParamOptions() enables you to set multiple values at one time for each bound parameter. This function allows the application to perform batch processing of the same SQL statement with one set of SQLPrepare(), SQLExecute(), and SQLBindParameter() calls.

Syntax

```
SQLRETURN   SQLParamOptions  (SQLHSTMT         hstmt,
                              SQLUINTEGER      crow,
                              SQLUINTEGER  FAR *pirow);
```

Function arguments

Table 158 lists the data type, use, and description for each argument in this function.

Table 158. SQLParamOptions() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Specifies a statement handle.
SQLUINTEGER	*crow*	input	Specifies the number of values for each parameter. If this value is greater than 1, then the *rgbValue* argument in SQLBindParameter() points to an array of parameter values, and the *pcbValue* argument points to an array of lengths.
SQLUINTEGER *	*pirow*	output (deferred)	Points to a buffer for the current parameter array index. As each set of parameter values is processed, this argument is set to the array index of that set. If a statement fails, this value can be used to determine how many statements were successfully processed. No value is returned if the *pirow* argument specifies a null pointer.

Usage

Use SQLParamOptions() to prepare a statement, and to execute that statement repeatedly for an array of parameter markers.

As a statement executes, the buffer to which the *pirow* argument points 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() return SQL_ERROR.

The output argument *pirow* points to a buffer that returns how many sets of parameters were successfully processed. If the statement that is processed is a query, *pirow* points to a buffer that returns the array index that is associated with the current result set, which returned by SQLMoreResults(). This value increments each time SQLMoreResults() is called.

Use the value in the buffer to which the *pirow* argument points for the following cases:

- When SQLParamData() returns SQL_NEED_DATA, use the value to determine which set of parameters need data.
- When SQLExecute() or SQLExecDirect() returns an error, use the value to determine which element in the parameter value array failed.
- When SQLExecute(), SQLExecDirect(), SQLParamData(), or SQLPutData() succeeds, the value is set to the value that the *crow* argument specifies to indicate that all elements of the array have been processed successfully.

Return codes

After you call SQLParamOptions(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 159 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 159. SQLParamOptions() SQLSTATEs

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY107	Row value out of range.	The value in the *crow* argument is less than 1.

Restrictions

Although this function is deprecated in ODBC 3.0, this function is not deprecated in DB2 ODBC. DB2 ODBC does not support the statement attributes that replace SQLParamOptions() in ODBC 3.0.

Example

See Figure 44 on page 417.

Related functions

The following functions relate to SQLParamOptions() calls. Refer to the descriptions of these functions for more information about how you can use SQLParamOptions() in your applications.

- "SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator" on page 85
- "SQLMoreResults() - Check for more result sets" on page 289
- "SQLSetStmtAttr() - Set statement attributes" on page 367

SQLPrepare() - Prepare a statement

Purpose

Table 160. SQLPrepare() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

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), SQLCloseCursor() must be called to close the cursor, before calling SQLPrepare().

Syntax

```
SQLRETURN   SQLPrepare      (SQLHSTMT         hstmt,
                              SQLCHAR      FAR  *szSqlStr,
                              SQLINTEGER        cbSqlStr);
```

Function arguments

Table 161 lists the data type, use, and description for each argument in this function.

Table 161. SQLPrepare() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle. There must not be an open cursor associated with *hstmt*.
SQLCHAR *	*szSqlStr*	input	SQL statement string.
SQLINTEGER	*cbSqlStr*	input	The length, in bytes, of the contents of the *szSqlStr* 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 nul-terminated.

Usage

If the SQL statement text contains vendor escape clause sequences, DB2 ODBC first modifies 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 465); then the SQL_NOSCAN statement attribute should be set to SQL_NOSCAN_ON at the statement level so that DB2 ODBC does not perform a scan for any vendor escape clauses.

When a statement is prepared using SQLPrepare(), the application can request information about the format of the result set (if the statement is a query) by calling:
- SQLNumResultCols()
- SQLDescribeCol()
- SQLColAttribute()

The SQL statement string can 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 question mark character (?) that indicates a position in the statement where an application supplied value is to be substituted when SQLExecute() is called. The bind parameter functions, SQLBindParameter() is 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 see “SQLExecute() - Execute a statement” on page 160.

After the application processes 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. SQLEndTran() must be called to issue COMMIT or ROLLBACK. For more information about SQL statements, that DB2 UDB for z/OS supports, see Table 1 on page 4.

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

After you call SQLPrepare(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 162 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 162. SQLPrepare() SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	*szSqlStr* contains an UPDATE or DELETE statement which did not contain a WHERE clause.
21S01	Insert value list does not match column list.	*szSqlStr* contains 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.	*szSqlStr* contains a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
34000	Invalid cursor name.	*szSqlStr* contains a positioned DELETE or a positioned UPDATE and the cursor referenced by the statement being executed is not open.

Table 162. SQLPrepare() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
37*xxx*[1]	Invalid SQL syntax.	*szSqlStr* contains one or more of the following: • A COMMIT • A ROLLBACK • An SQL statement that the connected database server cannot prepare • A statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belongs is rolled back due to deadlock or timeout.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
42*xxx* [1]	Syntax error or access rule violation	These SQLSTATEs indicate one of the following errors: • For **42**5*xx*, the authorization ID does not have permission to execute the SQL statement that the *szSqlStr* argument contains. • For **42***xxx*, a variety of syntax or access problems with the statement occur.
42S01	Database object already exists.	*szSqlStr* contains a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already exists.
42S02	Database object does not exist.	*szSqlStr* contains an SQL statement that references a table name or a view name that does not exist.
42S11	Index already exists.	*szSqlStr* contains a CREATE INDEX statement and the specified index name already exists.
42S12	Index not found.	*szSqlStr* contains a DROP INDEX statement and the specified index name does not exist.
42S21	Column already exists.	*szSqlStr* contains an ALTER TABLE statement and the column specified in the ADD clause is not unique or identifies an existing column in the base table.
42S22	Column not found.	*szSqlStr* contains an SQL statement that references a column name that does not exist.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	*szSqlStr* is a null pointer.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length.	The argument *cbSqlStr* is less than 1, but not equal to SQL_NTS.

Note:

1. *xxx* refers to any SQLSTATE with that class code. For example, **37***xxx* refers to any SQLSTATE with a class code of '**37**'.

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

Figure 26 shows an application that uses SQLPrepare() to prepare an SQL statement. This same SQL statement is then executed twice, each time with different parameter values.

```
 /******************************************************************/
 /*  Prepare a query and execute that query twice                  */
 /*  specifying a unique value for the parameter marker.           */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            *pDSN = "STLEC1";
   SWORD           cbCursor;
   SDWORD          cbValue1;
   SDWORD          cbValue2;
   char            employee [30];
   int             salary = 0;
   int             param_salary = 30000;

   char            *stmt = "SELECT NAME, SALARY FROM EMPLOYEE WHERE SALARY > ?";


   (void) printf ("**** Entering CLIP07.\n\n");

  /*****************************************************************/
  /* Allocate environment handle                                   */
  /*****************************************************************/

   rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);

   if (rc != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate connection handle to DSN                             */
  /*****************************************************************/

   rc = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);

   if (rc != SQL_SUCCESS )      // Could not get a connect handle
     goto dberror;
```

Figure 26. An application that prepares an SQL statement before execution (Part 1 of 5)

```
/******************************************************************/
/* CONNECT TO data source (STLEC1)                                */
/******************************************************************/

 rc = SQLConnect(hDbc,        // Connect handle
                     (SQLCHAR *) pDSN, // DSN
                     SQL_NTS,     // DSN is nul-terminated
                     NULL,        // Null UID
                     0   ,
                     NULL,        // Null Auth string
                     0);
 if (rc != SQL_SUCCESS )      // Connect failed
   goto dberror;

/******************************************************************/
/* Allocate statement handles                                     */
/******************************************************************/

rc = SQLAllocHandle (SQL_HANDLE_STMT, hDbc, &hStmt);

if (rc != SQL_SUCCESS)
  goto dberror;

/******************************************************************/
/* Prepare the query for multiple execution within current       */
/* transaction. Note that query is collapsed when transaction    */
/* is committed or rolled back.                                   */
/******************************************************************/

rc = SQLPrepare (hStmt,
                 (SQLCHAR *) stmt,
                 strlen(stmt));

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** PREPARE OF QUERY FAILED.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind employee name
                 1,
                 SQL_C_CHAR,
                 employee,
                 sizeof(employee),
                 &cbValue1);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF NAME FAILED.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind employee salary
                 2,
                 SQL_C_LONG,
                 &salary,
                 0,
                 &cbValue2);
if (rc != SQL_SUCCESS)
```

Figure 26. An application that prepares an SQL statement before execution (Part 2 of 5)

```
{
  (void) printf ("**** BIND OF SALARY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Bind parameter to replace '?' in query. This has an initial   */
/* value of 30000.                                               */
/*****************************************************************/

rc = SQLBindParameter (hStmt,
                       1,
                       SQL_PARAM_INPUT,
                       SQL_C_LONG,
                       SQL_INTEGER,
                       0,
                       0,
                       &param_salary,
                       0,
                       NULL);

/*****************************************************************/
/* Execute prepared statement to generate answer set.            */
/*****************************************************************/

rc = SQLExecute (hStmt);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** EXECUTE OF QUERY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Answer set is available -- Fetch rows and print employees     */
/* and salary.                                                   */
/*****************************************************************/

(void) printf ("**** Employees whose salary exceeds %d follow.\n\n",
               param_salary);

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Employee Name %s with salary %d.\n",
                 employee,
                 salary);
}

/*****************************************************************/
/* Close query --- note that query is still prepared. Then change*/
/* bound parameter value to 100000. Then re-execute query.       */
/*****************************************************************/

rc = SQLCloseCursor(hStmt);

param_salary = 100000;

rc = SQLExecute (hStmt);
if (rc != SQL_SUCCESS)
```

Figure 26. An application that prepares an SQL statement before execution (Part 3 of 5)

```
{
  (void) printf ("**** EXECUTE OF QUERY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Answer set is available -- Fetch rows and print employees     */
/* and salary.                                                   */
/*****************************************************************/

(void) printf ("**** Employees whose salary exceeds %d follow.\n\n",
               param_salary);

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Employee Name %s with salary %d.\n",
                 employee,
                 salary);
}

/*****************************************************************/
/* Deallocate statement handles -- statement is no longer in a   */
/* prepared state.                                               */
/*****************************************************************/

rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);

/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/

 rc = SQLDisconnect(hDbc);

 if (rc != SQL_SUCCESS)
   goto dberror;

/*****************************************************************/
/* Deallocate connection handle                                  */
/*****************************************************************/

 rc = SQLFreeHandle(SQL_HANDLE_DBC, hDbc);

 if (rc != SQL_SUCCESS)
   goto dberror;
```

Figure 26. An application that prepares an SQL statement before execution (Part 4 of 5)

```
  /*****************************************************************/
  /* Free environment handle                                       */
  /*****************************************************************/

   rc = SQLFreeHandle(SQL_HANDLE_ENV, hEnv);

   if (rc == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP07.\n\n");

   return RETCODE;
}
```

Figure 26. An application that prepares an SQL statement before execution (Part 5 of 5)

Related functions

The following functions relate to SQLPrepare() calls. Refer to the descriptions of these functions for more information about how you can use SQLPrepare() in your applications.

- "SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator" on page 85
- "SQLColAttribute() - Get column attributes" on page 101
- "SQLDescribeCol() - Describe column attributes" on page 131
- "SQLExecDirect() - Execute a statement directly" on page 154
- "SQLExecute() - Execute a statement" on page 160
- "SQLNumParams() - Get number of parameters in a SQL statement" on page 297
- "SQLNumResultCols() - Get number of result columns" on page 299
- "SQLSetParam() - Bind a parameter marker to a buffer" on page 364

SQLPrimaryKeys() - Get primary key columns of a table

Purpose

Table 163. SQLPrimaryKeys() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

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          hstmt,
                              SQLCHAR      FAR  *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR      FAR  *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR      FAR  *szTableName,
                              SQLSMALLINT       cbTableName);
```

Function arguments

Table 164 lists the data type, use, and description for each argument in this function.

Table 164. SQLPrimaryKeys() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLCHAR *	*szCatalogName*	input	Catalog qualifier of a three-part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	*cbCatalogName*	input	The length, in bytes, of *szCatalogName*.
SQLCHAR *	*szSchemaName*	input	Schema qualifier of table name.
SQLSMALLINT	*cbSchemaName*	input	The length, in bytes, of *szSchemaName*.
SQLCHAR *	*szTableName*	input	Table name.
SQLSMALLINT	*cbTableName*	input	The length, in bytes, of *szTableName*.

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 Table 165 on page 315, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME and ORDINAL_POSITION.

Because 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 bytes to be consistent with SQL92 limits. Because DB2 names are less than 128, you can always choose to set aside 128 characters (plus the nul-terminator) for the output buffer. Alternatively, you can call SQLGetInfo() with the *InfoType* argument set to each of the following values:

- SQL_MAX_CATALOG_NAME_LEN, to determine the length of TABLE_CAT columns that the connected DBMS supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected DBMS supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected DBMS supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected DBMS supports

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change. Table 165 lists each column in the result set this function generates.

Table 165. Columns returned by SQLPrimaryKeys()

Column number/name	Data type	Description
1 TABLE_CAT	VARCHAR(128)	This is always null.
2 TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
3 TABLE_NAME	VARCHAR(128) NOT NULL	Name of the specified table.
4 COLUMN_NAME	VARCHAR(128) NOT NULL	Primary key column name.
5 KEY_SEQ	SMALLINT NOT NULL	Column sequence number in the primary key, starting with 1.
6 PK_NAME	VARCHAR(128)	Primary key identifier. Contains a null value if not applicable to the data source.

The column names used by DB2 ODBC 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

After you call SQLPrimaryKeys(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 166 on page 316 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 166. SQLPrimaryKeys() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already open on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal SQL_NTS.
HYC00	Driver not capable.	DB2 ODBC does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

Figure 27 on page 317 shows an application that uses SQLPrimaryKeys() to locate a primary key for a table, and calls SQLColAttribute() to find the data type of the key.

```
/* ... */

#include <sqlcli1.h>

void main()
{
   SQLCHAR      rgbDesc_20];
   SQLCHAR      szTableName_20];
   SQLCHAR      szSchemaName_20];
   SQLCHAR      rgbValue_20];
   SQLINTEGER   pcbValue;
   SQLHENV      henv;
   SQLHDBC      hdbc;
   SQLHSTMT     hstmt;
   SQLSMALLINT  pscDesc;
   SQLINTEGER   pdDesc;
   SQLRETURN    rc;

   /*******************************************************************/
   /*   Initialization...                                             */
   /*******************************************************************/

   if( SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv)!= SQL_SUCCESS )
   {
       fprintf( stdout, "Error in SQLAllocHandle\n" );
       exit(1);
   }
   if( SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc)!= SQL_SUCCESS )
   {
       fprintf( stdout, "Error in SQLAllocHandle\n" );
       exit(1);
   }
   if( SQLConnect( hdbc,
                   NULL, SQL_NTS,
                   NULL, SQL_NTS,
                   NULL, SQL_NTS ) != SQL_SUCCESS )
   {
       fprintf( stdout, "Error in SQLConnect\n" );
       exit(1);
   }
   if( SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt)!= SQL_SUCCESS )
   {
       fprintf( stdout, "Error in SQLAllocHandle\n" );
       exit(1);
   }
```

Figure 27. An application that locates a table's primary key (Part 1 of 3)

```
/******************************************************************/
/*  Get primary key for table 'myTable' by using SQLPrimaryKeys    */
/******************************************************************/
rc = SQLPrimaryKeys( hstmt,
                     NULL, SQL_NTS,
                     (SQLCHAR*)szSchemaName, SQL_NTS,
                     (SQLCHAR*)szTableName, SQL_NTS );
if( rc != SQL_SUCCESS )
{
    goto exit;
}

/*
 *   Because all we need is the ordinal position, we'll bind column 5 from
 *   the result set.
 */
rc = SQLBindCol( hstmt,
                 5,
                 SQL_C_CHAR,
                 (SQLPOINTER)rgbValue,
                 20,
                 &pcbValue );
if( rc != SQL_SUCCESS )
{
    goto exit;
}
/*
 *   Fetch data...
 */
if( SQLFetch( hstmt ) != SQL_SUCCESS )
{
     goto exit;
}

/******************************************************************/
/*  Get data type for that column by calling SQLColAttribute().    */
/******************************************************************/

rc =  SQLColAttribute(  hstmt,
                        pcbValue,
                        SQL_COLUMN_TYPE,
                        rgbDesc,
                        20,
                        &pcbDesc,
                        &pfDesc );
if( rc != SQL_SUCCESS )
{
    goto exit;
}

/*
 *   Display the data type.
 */

fprintf( stdout, "Data type ==> %s\n", rgbDesc );
```

Figure 27. An application that locates a table's primary key (Part 2 of 3)

```
exit:
   /*******************************************************************/
   /* Clean up the environment...                                     */
   /*******************************************************************/

   SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);

   SQLDisconnect( hdbc );

   SQLFreeHandle(SQL_HANDLE_DBC, hdbc);

   SQLFreeHandle(SQL_HANDLE_ENV, henv);

}
```

Figure 27. An application that locates a table's primary key (Part 3 of 3)

Related functions

The following functions relate to SQLPrimaryKeys() calls. Refer to the descriptions of these functions for more information about how you can use SQLPrimaryKeys() in your applications.

- "SQLForeignKeys() - Get a list of foreign key columns" on page 178
- "SQLStatistics() - Get index and statistics information for a base table" on page 381

SQLProcedureColumns() - Get procedure input/output parameter information

Purpose

Table 167. SQLProcedureColumns() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

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          hstmt,
                              SQLCHAR      FAR  *szProcCatalog,
                              SQLSMALLINT       cbProcCatalog,
                              SQLCHAR      FAR  *szProcSchema,
                              SQLSMALLINT       cbProcSchema,
                              SQLCHAR      FAR  *szProcName,
                              SQLSMALLINT       cbProcName,
                              SQLCHAR      FAR  *szColumnName,
                              SQLSMALLINT       cbColumnName);
```

Function arguments

Table 168 lists the data type, use, and description for each argument in this function.

Table 168. SQLProcedureColumns() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLCHAR *	*szProcCatalog*	input	Catalog qualifier of a three-part procedure name. This must be a null pointer or a zero length string.
SQLSMALLINT	*cbProcCatalog*	input	The length, in bytes, of *szProcCatalog*. This must be set to 0.
SQLCHAR *	*szProcSchema*	input	Buffer that can contain a *pattern-value* to qualify the result set by schema name. If you do not want to qualify the result set by schema name, use a null pointer or a zero length string for this argument.
SQLSMALLINT	*cbProcSchema*	input	The length, in bytes, of *szProcSchema*.
SQLCHAR *	*szProcName*	input	Buffer that can contain a *pattern-value* to qualify the result set by procedure name. If you do not want to qualify the result set by procedure name, use a null pointer or a zero length string for this argument.
SQLSMALLINT	*cbProcName*	input	The length, in bytes, of *szProcName*.

Table 168. SQLProcedureColumns() arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*szColumnName*	input	Buffer that can 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 *szProcName* and/or *szProcSchema*. If you do not want to qualify the result set by parameter name, use a null pointer or a zero length string for this argument.
SQLSMALLINT	*cbColumnName*	input	The length, in bytes, of *szColumnName*.

For more information about valid search patterns, see "Querying catalog information" on page 407.

Usage

Registered stored procedures are defined in the SYSIBM.SYSROUTINES catalog table. For servers that do not provide facilities for a stored procedure catalog, this function returns an empty result set.

DB2 ODBC returns 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. Table 169 on page 322 lists the columns in the result set.

Because 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 bytes to be consistent with SQL92 limits. Because DB2 names are less than 128 bytes, the application can choose to always set aside 128 bytes (plus the nul-terminator) for the output buffer. Alternatively, you can call SQLGetInfo() with the *InfoType* argument set to each of the following values:

- SQL_MAX_CATALOG_NAME_LEN, to determine the length of TABLE_CAT columns that the connected DBMS supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected DBMS supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected DBMS supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected DBMS supports

Applications should be aware that columns beyond the last column might be defined in future releases. Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change. Table 169 on page 322 lists these columns.

Table 169. Columns returned by SQLProcedureColumns()

Column number/name	Data type	Description
1 PROCEDURE_CAT	VARCHAR(128)	The is always null.
2 PROCEDURE_SCHEM	VARCHAR(128)	The name of the schema containing PROCEDURE_NAME. (This is also NULL for DB2 UDB for z/OS SQLProcedureColumns() result sets.)
3 PROCEDURE_NAME	VARCHAR(128)	Name of the procedure.
4 COLUMN_NAME	VARCHAR(128)	Name of the parameter.
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.[1] • 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.[1] • SQL_RESULT_COL: this parameter is actually a column in the result set.[1] **Requirement:** For SQL_PARAM_OUTPUT and SQL_RETURN_VALUE support, you must have ODBC 2.0 or higher. **Note:** 1. These values are not returned.
6 DATA_TYPE	SMALLINT NOT NULL	SQL data type.
7 TYPE_NAME	VARCHAR(128) NOT NULL	Character string representing the name of the data type corresponding to DATA_TYPE.
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 Table 234 on page 509.
9 BUFFER_LENGTH	INTEGER	The maximum number of bytes for the associated C buffer to store data from this parameter if SQL_C_DEFAULT is specified on the SQLBindCol(), SQLGetData() and SQLBindParameter() calls. This length excludes any nul-terminator. For exact numeric data types, the length accounts for the decimal and the sign. See Table 236 on page 511.

Table 169. Columns returned by SQLProcedureColumns() (continued)

Column number/name	Data type	Description
10 DECIMAL_DIGITS	SMALLINT	The scale of the parameter. NULL is returned for data types where scale is not applicable. See Table 235 on page 510.
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.
12 NULLABLE	SMALLINT NOT NULL	SQL_NO_NULLS if the parameter does not accept NULL values. SQL_NULLABLE if the parameter accepts NULL values.
13 REMARKS	VARCHAR(254)	Might contain descriptive information about the parameter.
14 COLUMN_DEF	VARCHAR(254)	The column's default value. If the default value is: • A numeric literal, this column contains the character representation of the numeric literal with no enclosing single quotes. • A character string, this column is that string enclosed in single quotes. • A pseudo-literal, such as for DATE, TIME, and TIMESTAMP columns, this column contains the keyword of the pseudo-literal (for example, CURRENT DATE) with no enclosing single quotes. • NULL, this column returns the word NULL, with no enclosing single quotes. If the default value cannot be represented without truncation, this column contains TRUNCATED with no enclosing single quotes. If no default value is specified, this column is NULL.
15 SQL_DATA_TYPE	SMALLINT NOT NULL	The SQL data type. This columns is the same as the DATA_TYPE column. For datetime data types, the SQL_DATA_TYPE field in the result set is SQL_DATETIME, and the SQL_DATETIME_SUB field returns the subcode for the specific datetime data type (SQL_CODE_DATE, SQL_CODE_TIME or SQL_CODE_TIMESTAMP).
16 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 a null value.
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 value.

Table 169. Columns returned by SQLProcedureColumns() (continued)

Column number/name	Data type	Description
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 provided on the CALL statement. The leftmost argument has an ordinal position of 1.
19 IS_NULLABLE	VARCHAR(128)	One of the following: • ″NO″, if the column does not include null values • ″YES″, if the column can include null values • Zero-length string if nullability is unknown. The value returned for this column is different than the value returned for the NULLABLE column. (See the description of the NULLABLE column.)

The column names used by DB2 ODBC 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

After you call SQLProcedureColumns(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 170 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 170. SQLProcedureColumns() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is opened on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
42601	PARMLIST syntax error.	The PARMLIST value in the stored procedures catalog table contains a syntax error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length	The value of one of the name length arguments is less than 0, but not equal SQL_NTS.

Table 170. SQLProcedureColumns() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HYC00	Driver not capable.	This SQLSTATE is returned for one or more of the following reasons: • DB2 ODBC does not support *catalog* as a qualifier for procedure name. • The connected server does not support *schema* as a qualifier for procedure name.

Restrictions

SQLProcedureColumns() does not return information about the attributes of result sets that stored procedures can return.

If an application is connected to a DB2 server that does not provide support for stored procedures, or for a stored procedure catalog, SQLProcedureColumns() returns an empty result set.

Example

Figure 28 on page 326 shows an application that retrieves input, input/output, and output parameters associated with a procedure.

```
 /******************************************************************/
 /*  Invoke SQLProcedureColumns and enumerate all rows retrieved.  */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            *pDSN = "STLEC1";
   char            procedure_name [20];
   char            parameter_name [20];
   char            ptype          [20];
   SQLSMALLINT     parameter_type = 0;
   SQLSMALLINT     data_type = 0;
   char            type_name      [20];
   SWORD           cbCursor;
   SDWORD          cbValue3;
   SDWORD          cbValue4;
   SDWORD          cbValue5;
   SDWORD          cbValue6;
   SDWORD          cbValue7;
   char            ProcCatalog [20] = {0};
   char            ProcSchema  [20] = {0};
   char            ProcName    [20] = {"DOIT%"};
   char            ColumnName  [20] = {"P%"};
   SQLSMALLINT     cbProcCatalog = 0;
   SQLSMALLINT     cbProcSchema  = 0;
   SQLSMALLINT     cbProcName    = strlen(ProcName);
   SQLSMALLINT     cbColumnName  = strlen(ColumnName);
```

Figure 28. An application that retrieves parameters associated with a procedure (Part 1 of 5)

```
 (void) printf ("**** Entering CLIP12.\n\n");

/*****************************************************************/
/* Allocate environment handle                                   */
/*****************************************************************/

 RETCODE = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;

/*****************************************************************/
/* Allocate connection handle to DSN                             */
/*****************************************************************/

 RETCODE = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);

 if( RETCODE != SQL_SUCCESS )      // Could not get a connect handle
   goto dberror;

/*****************************************************************/
/* CONNECT TO data source (STLEC1)                               */
/*****************************************************************/

 RETCODE = SQLConnect(hDbc,        // Connect handle
                      (SQLCHAR *) pDSN, // DSN
                      SQL_NTS,     // DSN is nul-terminated
                      NULL,        // Null UID
                      0   ,
                      NULL,        // Null Auth string
                      0);

 if( RETCODE != SQL_SUCCESS )      // Connect failed
   goto dberror;

/*****************************************************************/
/* Allocate statement handles                                    */
/*****************************************************************/

rc = SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt);

if (rc != SQL_SUCCESS)
  goto exit;

/*****************************************************************/
/* Invoke SQLProcedureColumns and retrieve all rows within       */
/* answer set.                                                   */
/*****************************************************************/

rc = SQLProcedureColumns (hStmt                ,
                          (SQLCHAR *) ProcCatalog,
                          cbProcCatalog        ,
                          (SQLCHAR *) ProcSchema ,
                          cbProcSchema         ,
                          (SQLCHAR *) ProcName   ,
                          cbProcName           ,
                          (SQLCHAR *) ColumnName ,
                          cbColumnName);
```

Figure 28. An application that retrieves parameters associated with a procedure (Part 2 of 5)

```
if (rc != SQL_SUCCESS)
{
  (void) printf ("**** SQLProcedureColumns Failed.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind procedure_name
                 3,
                 SQL_C_CHAR,
                 procedure_name,
                 sizeof(procedure_name),
                 &cbValue3);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of procedure_name Failed.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind parameter_name
                 4,
                 SQL_C_CHAR,
                 parameter_name,
                 sizeof(parameter_name),
                 &cbValue4);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of parameter_name Failed.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind parameter_type
                 5,
                 SQL_C_SHORT,
                 &parameter_type,
                 0,
                 &cbValue5);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of parameter_type Failed.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind SQL data type
                 6,
                 SQL_C_SHORT,
                 &data_type,
                 0,
                 &cbValue6);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of data_type Failed.\n");
  goto dberror;
}
```

Figure 28. An application that retrieves parameters associated with a procedure (Part 3 of 5)

```
rc = SQLBindCol (hStmt,           // bind type_name
                 7,
                 SQL_C_CHAR,
                 type_name,
                 sizeof(type_name),
                 &cbValue7);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of type_name Failed.\n");
  goto dberror;
}

/*****************************************************************/
/* Answer set is available - Fetch rows and print parameters for */
/* all procedures.                                               */
/*****************************************************************/

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Procedure Name = %s. Parameter %s",
                 procedure_name,
                 parameter_name);

  switch (parameter_type)
  {
    case SQL_PARAM_INPUT        :
      (void) strcpy (ptype, "INPUT");
      break;
    case SQL_PARAM_OUTPUT       :
      (void) strcpy (ptype, "OUTPUT");
      break;
    case SQL_PARAM_INPUT_OUTPUT :
      (void) strcpy (ptype, "INPUT/OUTPUT");
      break;
    default                     :
      (void) strcpy (ptype, "UNKNOWN");
      break;
  }

  (void) printf (" is %s. Data Type is %d. Type Name is %s.\n",
                 ptype     ,
                 data_type ,
                 type_name);
}

/*****************************************************************/
/* Deallocate statement handles -- statement is no longer in a   */
/* prepared state.                                               */
/*****************************************************************/

rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);

/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/

 RETCODE = SQLDisconnect(hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
```

Figure 28. An application that retrieves parameters associated with a procedure (Part 4 of 5)

```
  /*****************************************************************/
  /* Deallocate connection handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeHandle(SQL_HANDLE_DBC, hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Free Environment Handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeHandle(SQL_HANDLE_ENV, hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP12.\n\n");

   return RETCODE;
}
```

Figure 28. An application that retrieves parameters associated with a procedure (Part 5 of 5)

Related functions

The following functions relate to SQLProcedureColumns() calls. Refer to the descriptions of these functions for more information about how you can use SQLProcedureColumns() in your applications.

- "SQLProcedures() - Get a list of procedure names" on page 331

SQLProcedures() - Get a list of procedure names

Purpose

Table 171. SQLProcedures() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

SQLProcedures() returns a list of procedure names that have been registered at the server, and that 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          hstmt,
                              SQLCHAR     FAR   *szProcCatalog,
                              SQLSMALLINT       cbProcCatalog,
                              SQLCHAR     FAR   *szProcSchema,
                              SQLSMALLINT       cbProcSchema,
                              SQLCHAR     FAR   *szProcName,
                              SQLSMALLINT       cbProcName);
```

Function arguments

Table 172 lists the data type, use, and description for each argument in this function.

Table 172. SQLProcedures() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLCHAR *	*szProcCatalog*	input	Catalog qualifier of a three-part procedure name. This must be a null pointer or a zero length string.
SQLSMALLINT	*cbProcCatalog*	input	The length, in bytes, of *szProcCatalog*. This must be set to 0.
SQLCHAR *	*szProcSchema*	input	Buffer that can contain a *pattern-value* to qualify the result set by schema name. If you do not want to qualify the result set by schema name, use a null pointer or a zero length string for this argument.
SQLSMALLINT	*cbProcSchema*	input	The length, in bytes, of *szProcSchema*.
SQLCHAR *	*szProcName*	input	Buffer that can contain a *pattern-value* to qualify the result set by table name. If you do not want to qualify the result set by table name, use a null pointer or a zero length string for this argument.
SQLSMALLINT	*cbProcName*	input	The length, in bytes, of *szProcName*.

For more information about valid search patterns, see "Querying catalog information" on page 407.

Usage

| Registered stored procedures are defined in the SYSIBM.SYSROUTINES catalog
| table. For servers that do not provide facilities for a stored procedure catalog, this
| function returns an empty result set.

The result set returned by SQLProcedures() contains the columns that are listed in Table 173 in the order given. The rows are ordered by PROCEDURE_CAT, PROCEDURE_SCHEM, and PROCEDURE_NAME.

Because 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 bytes to be consistent with SQL92 limits. Because DB2 names are less than 128 bytes, the application can choose to always set aside 128 bytes (plus the nul-terminator) for the output buffer. Alternatively, you can call SQLGetInfo() with the *InfoType* argument set to each of the following values:

- SQL_MAX_CATALOG_NAME_LEN, to determine the length of TABLE_CAT columns that the connected DBMS supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected DBMS supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected DBMS supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected DBMS supports

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change. Table 174 on page 333 lists these columns.

Table 173. Columns returned by SQLProcedures()

Column number	Column name	Data type	Description
1	PROCEDURE_CAT	VARCHAR(128)	This is always null.
2	PROCEDURE_SCHEM	VARCHAR(128)	The name of the schema containing PROCEDURE_NAME.
3	PROCEDURE_NAME	VARCHAR(128) NOT NULL	The name of the procedure.
4	NUM_INPUT_PARAMS	INTEGER not NULL	Number of input parameters.
5	NUM_OUTPUT_PARAMS	INTEGER not NULL	Number of output parameters.
6	NUM_RESULT_SETS	INTEGER not NULL	Number of result sets returned by the procedure.
7	REMARKS	VARCHAR(254)	Contains the descriptive information about the procedure.

Table 173. Columns returned by SQLProcedures() (continued)

Column number	Column name	Data type	Description
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 ODBC always returns SQL_PT_PROCEDURE.

The column names used by DB2 ODBC 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

After you call SQLProcedures(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 174 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 174. SQLProcedures() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is opened on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal to SQL_NTS.
HYC00	Driver not capable.	This SQLSTATE is returned for one or more of the following reasons: • DB2 ODBC does not support *catalog* as a qualifier for procedure name. • The connected server does not supported schema as a qualifier for procedure name.

Restrictions

If an application is connected to a DB2 server that does not provide support for stored procedures, or for a stored procedure catalog, SQLProcedureColumns() returns an empty result set.

Example

Figure 29 shows an application that prints a list of procedures registered at the server. The application uses SQLProcedures() retrieve these procedures and to establish a search pattern.

```
/* ... */

    printf("Enter Procedure Schema Name Search Pattern:\n");
    gets(proc_schem.s);

    rc = SQLProcedures(hstmt, NULL, 0, proc_schem.s, SQL_NTS, "%", SQL_NTS);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) proc_schem.s, 129,
                    &proc_schem.ind);

    rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) proc_name.s, 129,
                    &proc_name.ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) remarks.s, 255,
                    &remarks.ind);

    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 */
/* ... */
```

Figure 29. An application that prints a list of registered procedures

Related functions

The following functions relate to SQLProcedures() calls. Refer to the descriptions of these functions for more information about how you can use SQLProcedures() in your applications.

- “SQLProcedureColumns() - Get procedure input/output parameter information” on page 320

SQLPutData() - Pass a data value for a parameter

Purpose

Table 175. SQLPutData() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLPutData() is called following an SQLParamData() call returning SQL_NEED_DATA to supply a parameter data value. 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         hstmt,
                              SQLPOINTER       rgbValue,
                              SQLINTEGER       cbValue);
```

Function arguments

Table 176 lists the data type, use, and description for each argument in this function.

Table 176. SQLPutData() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLPOINTER	*rgbValue*	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	*cbValue*	input	The length, in bytes, of *rgbValue*. 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 *cbValue*. *cbValue* 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 *rgbValue* buffer.

Usage

For a description on the SQLParamData() and SQLPutData() sequence, see "Sending or retrieving long data values in pieces" on page 412.

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 using 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 fail. In addition, all function calls referencing the parent *hdbc* of *hstmt* 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:
- SQLAllocHandle()
- SQLSetConnectAttr()
- SQLNativeSql()
- SQLEndTran()

If they are invoked during an SQL_NEED_DATA sequence, these functions return SQL_ERROR with SQLSTATE of **HY**010 and the processing of the SQL_DATA_AT_EXEC parameters is not affected.

If one or more calls to SQLPutData() for a single parameter results in SQL_SUCCESS, attempting to call SQLPutData() with *cbValue* 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

After you call SQLPutData(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Some of the following diagnostic conditions are also reported on the final SQLParamData() call rather than at the time the SQLPutData() is called. Table 177 lists each SQLSTATE with a description and explanation for each value.

Table 177. SQLPutData() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	This SQLSTATE is returned for one or more of the following reasons: • The data sent for a numeric parameter is truncated without the loss of significant digits. • Timestamp data sent for a date or time column is truncated. (SQLPutData() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)

Table 177. SQLPutData() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22001	String data right truncation.	More data is sent for a binary or char data than the data source can support for that column.
22008	Invalid datetime format or datetime field overflow.	The data value sent for a date, time, or timestamp parameters is invalid.
22018	Error in assignment.	The data sent for a parameter is incompatible with the data type of the associated table column.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	The argument *rgbValue* is a NULL pointer, and the argument *cbValue* is neither 0 nor SQL_NULL_DATA.
HY010	Function sequence error.	The statement handle *hstmt* must be in a need data state and must have been positioned on an SQL_DATA_AT_EXEC parameter using a previous SQLParamData() call.
HY019	Numeric value out of range.	This SQLSTATE is returned for one or more of the following reasons: • The data sent for a numeric parameter causes the whole part of the number to be truncated when it is assigned to the associated column. • SQLPutData() is called more than once for a fixed-length parameter.
HY090	Invalid string or buffer length.	The argument *rgbValue* is not a null pointer, and the argument *cbValue* is less than 0, but not equal to SQL_NTS or SQL_NULL_DATA.

Restrictions

A new value for *pcbValue*, 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. Because the concept of default values does not apply to DB2 stored procedure arguments, specification of this value for the *pcbValue* argument results in an error when the CALL statement is executed because the SQL_DEFAULT_PARAM value is considered an invalid length.

ODBC 2.0 also introduced the SQL_LEN_DATA_AT_EXEC(*length*) macro to be used with the *pcbValue* argument. The macro is used to specify the sum total length, in bytes, of the entire data that would be sent for character or binary C data using the subsequent SQLPutData() calls. Because the DB2 ODBC driver does not need this information, the macro is not needed. To check if the driver needs this information, call SQLGetInfo() with the *InfoType* argument set to SQL_NEED_LONG_DATA_LEN. The DB2 ODBC driver returns ’N’ to indicate that this information is not needed by SQLPutData().

Example

See Figure 19 on page 213.

Related functions

The following functions relate to SQLPutData() calls. Refer to the descriptions of these functions for more information about how you can use SQLPutData() in your applications.

- "SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator" on page 85
- "SQLCancel() - Cancel statement" on page 97
- "SQLExecute() - Execute a statement" on page 160
- "SQLExecDirect() - Execute a statement directly" on page 154
- "SQLParamData() - Get next parameter for which a data value is needed" on page 301

SQLRowCount() - Get row count

Purpose

Table 178. SQLRowCount() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

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         hstmt,
                              SQLINTEGER  FAR   *pcrow);
```

Function arguments

Table 179 lists the data type, use, and description for each argument in this function.

Table 179. SQLRowCount() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLINTEGER *	*pcrow*	output	Pointer to location where the number of rows affected is stored.

Usage

If the last executed statement referenced by the input statement handle is not an UPDATE, INSERT, or DELETE statement, or if it did not execute successfully, then the function sets the contents of *pcrow* to -1.

If SQLRowCount() is executed after the SQLExecDirect() or SQLExecute() of an SQL statement other than INSERT, UPDATE, or DELETE, it results in return code 0 and *pcrow* is set to -1.

Any rows in other tables that might be affected by the statement (for example, cascading deletes) are not included in the count.

If SQLRowCount() is executed after a built-in function (for example, SQLTables()), it results in return code -1 and SQLSTATE **HY**010.

Return codes

After you call SQLRowCount(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 180 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 180. SQLRowCount() SQLSTATEs

SQLSTATE	Description	Explanation
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called prior to calling SQLExecute() or SQLExecDirect() for the *hstmt*.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.

Restrictions

None.

Example

See Figure 13 on page 134.

Related functions

The following functions relate to SQLRowCount() calls. Refer to the descriptions of these functions for more information about how you can use SQLRowCount() in your applications.

- “SQLExecDirect() - Execute a statement directly” on page 154
- “SQLExecute() - Execute a statement” on page 160
- “SQLNumResultCols() - Get number of result columns” on page 299

SQLSetColAttributes() - Set column attributes

Purpose

Table 181. SQLSetColAttributes() specifications

ODBC	X/OPEN CLI	ISO CLI
No	No	No

SQLSetColAttributes() sets the data source result descriptor (column name, type, precision, scale and nullability) for one column in the result set so that the DB2 ODBC implementation does not need to obtain the descriptor information from the DBMS server.

Syntax

```
SQLRETURN  SQLSetColAttributes (SQLHSTMT        hstmt,
                               SQLUSMALLINT     icol,
                               SQLCHAR     FAR  *pszColName,
                               SQLSMALLINT      cbColName,
                               SQLSMALLINT      fSqlType,
                               SQLUINTEGER      cbColDef,
                               SQLSMALLINT      ibScale,
                               SQLSMALLINT      fNullable);
```

Function arguments

Table 182 lists the data type, use, and description for each argument in this function.

Table 182. SQLSetColAttributes() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLUSMALLINT	*icol*	input	Column number of result data, ordered sequentially left to right, starting at 1.
SQLCHAR *	*szColName*	input	Pointer to the column name. If the column is unnamed or is an expression, this pointer can be set to NULL, or an empty string can be used.
SQLSMALLINT	*cbColName*	input	The length, in bytes, of *szColName* buffer.

Table 182. SQLSetColAttributes() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*fSqlType*	input	The SQL data type of the column. The following values are recognized: • SQL_BINARY • SQL_BLOB • SQL_CHAR • SQL_CLOB • SQL_DBCLOB • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_ROWID • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC
SQLUINTEGER	*cbColDef*	input	The precision of the column on the data source.
SQLSMALLINT	*ibScale*	input	The scale of the column on the data source. This is ignored for all data types except SQL_DECIMAL, SQL_NUMERIC, SQL_TYPE_TIMESTAMP.
SQLSMALLINT	*fNullable*	input	Indicates whether the column allows null values. This must of one of the following values: • SQL_NO_NULLS - the column does not allow null values. • SQL_NULLABLE - the column allows null values.

Usage

This function is designed to help reduce the amount of network traffic that can result when an application is fetching result sets that contain an extremely large number of columns. If the application has advanced knowledge of the characteristics of the descriptor information of a result set (that is, the exact number of columns, column name, data type, nullability, precision, or scale), then it can inform DB2 ODBC rather than having DB2 ODBC obtain this information from the database, thus reducing the quantity of network traffic.

An application typically calls SQLSetColAttributes() after a call to SQLPrepare() and before the associated call to SQLExecute(). An application can also call SQLSetColAttributes() before a call to SQLExecDirect(). This function is valid only after the statement attribute SQL_NODESCRIBE has been set to SQL_NODESCRIBE_ON for this statement handle.

SQLSetColAttributes() informs DB2 ODBC of the column name, type, and length that would be generated by the subsequent execution of the query. This information allows DB2 ODBC to determine whether any data conversion is necessary when the result is returned to the application.

Recommendation: Use this function only if you know the exact nature of the result set.

The application must provide the result descriptor information for every column in the result set or an error occurs on the subsequent fetch (SQLSTATE **07**002). Using this function only benefits those applications that handle an extremely large number (hundreds) of columns in a result set, otherwise the effect is minimal.

Return codes

After you call SQLSetColAttributes(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 183 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 183. SQLSetColAttributes() SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The *szColName* argument contains a column name that is too long. To obtain the maximum length of the column name, call SQLGetInfo with the *InfoType* SQL_MAX_COLUMN_NAME_LEN.
24000	Invalid cursor state.	A cursor is open on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY000	General error.	An error occurred for which there is no specific SQLSTATE and for which no implementation defined SQLSTATE is defined. The error message returned by SQLGetDiagRec() in the argument *szErrorMsg* describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY004	Invalid SQL data type.	The value specified for the argument *fSqlType* is not a valid SQL data type.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument *cbColName* is less than 0 and not equal to SQL_NTS.
HY099	Nullable type out of range.	The value specified for *fNullable* is invalid.

Table 183. SQLSetColAttributes() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY104	Invalid precision or scale value.	This SQLSTATE is returned for one or more of the following reasons: • The value specified for *fSqlType* is either SQL_DECIMAL or SQL_NUMERIC and the value specified for *cbColDef* is less than 1, or the value specified for *ibScale* is less than 0 or greater than the value for the argument *cbColDef* (precision). • The value specified for *fSqlType* is SQL_TYPE_TIMESTAMP and the value for *ibScale* is less than 0 or greater than 6.
HY002	Invalid column number.	The value specified for the argument *icol* is less than 1 or greater than the maximum number of columns supported by the server.

Restrictions

None.

Example

Figure 30 on page 345 shows an application that uses SQLSetColAttributes() to set the data source results descriptor.

```
/* ... */
    SQLCHAR         stmt[] =
    { "Select id, name from staff" };
/* ... */

  /* Tell DB2 ODBC not to get column attribute from the server for this hstmt */
    rc = SQLSetStmtAttr(hstmt,SQL_ATTR,NODESCRIBE,(void *)SQL_NODESCRIBE_ON, 0);

    rc = SQLPrepare(hstmt, stmt, SQL_NTS);

/* Provide the columns attributes to DB2 ODBC for this hstmt */
    rc = SQLSetColAttributes(hstmt, 1, "-ID-", SQL_NTS, SQL_SMALLINT,
                             5, 0, SQL_NO_NULLS);
    rc = SQLSetColAttributes(hstmt, 2, "-NAME-", SQL_NTS, SQL_CHAR,
                             9, 0, SQL_NULLABLE);
    rc = SQLExecute(hstmt);

    print_results(hstmt); /* Call sample function to print column attributes
                             and fetch and print rows.  */

    rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);

    rc = SQLEndTran(SQL_HANDLE, DBC, hdbc, SQL_COMMIT);

    printf("Disconnecting .....\n");
    rc = SQLDisconnect(hdbc);

    rc = SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));

    rc = SQLFreeHandle(SQL_HANDLE_ENV, henv);
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));

    return (SQL_SUCCESS);
}                               /* end main */
```

Figure 30. An application that sets the data source results descriptor

Related functions

The following functions relate to SQLSetColAttributes() calls. Refer to the descriptions of these functions for more information about how you can use SQLSetColAttributes() in your applications.

- “SQLColAttribute() - Get column attributes” on page 101
- “SQLDescribeCol() - Describe column attributes” on page 131
- “SQLExecute() - Execute a statement” on page 160
- “SQLExecDirect() - Execute a statement directly” on page 154
- “SQLPrepare() - Prepare a statement” on page 306

SQLSetConnectAttr() - Set connection attributes

Purpose

Table 184. SQLSetConnectAttr() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLSetConnectAttr() sets attributes that govern aspects of connections.

Syntax

```
SQLRETURN  SQLSetConnectAttr (SQLHDBC          ConnectionHandle,
                              SQLINTEGER       Attribute,
                              SQLPOINTER       ValuePtr,
                              SQLINTEGER       StringLength);
```

Function arguments

Table 185 lists the data type, use, and description for each argument in this function.

Table 185. SQLSetConnectAttr() arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle.
SQLINTEGER	*Attribute*	input	Connection attribute to set. Refer to Table 187 on page 347 for a complete list of attributes.
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 nul-terminated character string. If the *Attribute* argument is a driver-specific value, the value in **ValuePtr* might be a signed integer.
SQLINTEGER	*StringLength*	input	Information about the **ValuePtr* argument. • For ODBC-defined attributes: – If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*. – If *ValuePtr* points to an integer, *BufferLength* is ignored. • For driver-defined attributes (IBM extension): – If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr* or SQL_NTS if it is a nul-terminated string. – If *ValuePtr* points to an integer, *BufferLength* is ignored.

Usage

SQLSetConnectAttr() sets attributes that govern aspects of connections.

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 or after a connection is made. Other attributes cannot be set after a statement is allocated. Table 186 indicates when each of the connection attributes can be set.

Table 186. When connection attributes can be set

Attribute	**Before connection**	**After connection**	**After statements allocated**
SQL_ATTR_ACCESS_MODE	Yes	Yes	Yes[1]
SQL_ATTR_AUTOCOMMIT	Yes	Yes	Yes[2]
SQL_ATTR_CONNECTTYPE	Yes	No	No
SQL_ATTR_CURRENT_SCHEMA	Yes	Yes	Yes
SQL_ATTR_MAXCONN	Yes	No	No
SQL_ATTR_SYNC_POINT	Yes	No	No
SQL_ATTR_TXN_ISOLATION	No	Yes	Yes

Notes:

1. Attribute only affects subsequently allocated statements.
2. Attribute can be set only if all transactions on the connections are closed.

Table 187 lists the SQLSetConnectAttr() *Attribute* values. Values shown in **bold** are default values unless they are otherwise specified in the ODBC initialization file. DB2 ODBC supports all of the ODBC 2.0 *Attribute* values that are renamed in ODBC 3.0.

For a summary of the *Attribute* values renamed in ODBC 3.0, see Table 256 on page 526.

ODBC applications that need to set statement attributes should use SQLSetStmtAttr(). The ability to set statement attributes on the connect level is supported, but it is not recommended.

Table 187. Connection attributes

Attribute	*ValuePtr*
SQL_ATTR_ACCESS_MODE	A 32-bit integer value which can be either: • SQL_MODE_READ_ONLY: Indicates that the application is not 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 ODBC does not ensure that requests to the database are *read-only*. If an update request is issued, DB2 ODBC processes it using the transaction isolation level it selected as a result of the SQL_MODE_READ_ONLY setting. • **SQL_MODE_READ_WRITE**: Indicates that the application is making updates on data from this point on. DB2 ODBC goes back to using the default transaction isolation level for this connection. SQL_MODE_READ_WRITE is the default. This connection must have no outstanding transactions.

Table 187. Connection attributes (continued)

Attribute	*ValuePtr*
SQL_ATTR_AUTOCOMMIT[1]	A 32-bit integer value that specifies whether to use autocommit or manual commit mode: • SQL_AUTOCOMMIT_OFF: The application must manually, explicitly commit or rollback transactions with SQLEndTran() calls. • **SQL_AUTOCOMMIT_ON**: DB2 ODBC operates in autocommit 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. This is the default value. **Exception:** If the connection is a coordinated distributed unit of work connection, the default is **SQL_AUTOCOMMIT_OFF**. When specifying autocommit, the application can have only one outstanding statement per connection. For example, two cursors cannot be open, otherwise unpredictable results can occur. An open cursor must be closed before another query is executed. Because in many DB2 environments the execution of the SQL statements and the commit can be flowed separately to the database server, autocommit can be expensive. The application developer should take this into consideration when selecting the autocommit mode. Changing from manual-commit to autocommit mode commits any open transaction on the connection. For information about setting this attribute see “Disabling autocommit” on page 475.

Table 187. Connection attributes (continued)

Attribute	*ValuePtr*
SQL_ATTR_CONNECTTYPE[2]	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 attribute must be considered in conjunction with the SQL_ATTR_SYNC_POINT connection attribute. The possible values are: • **SQL_CONCURRENT_TRANS**: The application can have concurrent multiple connections to any one database or to multiple databases. This attribute value corresponds to the specification of the type 1 CONNECT in embedded SQL. Each connection has its own commit scope. No effort is made to enforce coordination of transaction. 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 attribute value 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 attribute. In contrast to the SQL_CONCURRENT_TRANS setting described above, the application is permitted only one open connection per database. **Important:** This connection type results in the default for SQL_ATTR_AUTOCOMMIT connection attribute to be SQL_AUTOCOMMIT_OFF. This attribute must be set before making a connect request; otherwise, the SQLSetConnectAttr() call is 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. **IBM specific:** This attribute is an IBM-defined extension. **Recommendation:** Have 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.
SQL_ATTR_CURRENT_SCHEMA	A nul-terminated character string containing the name of the schema to be used by DB2 ODBC for the SQLColumns() call if the *szSchemaName* pointer is set to null. To reset this attribute, specify this attribute with a zero length or a null pointer for the *vParam* argument. This attribute 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 attribute can be set at any time and is effective on the next SQLColumns() call where the *szSchemaName* pointer is null. **IBM specific:** This attribute is an IBM-defined extension.

Table 187. Connection attributes (continued)

Attribute	*ValuePtr*
SQL_ATTR_MAXCONN[3]	A 32-bit integer value corresponding to the number of maximum concurrent connections that an application wants 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. The value that is in effect when the first connection is established is the value that is used. When the first connection is established, attempts to change this value are rejected. **IBM specific:** This attribute is an IBM-defined extension. **Recommendation:** Have 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 because SQLSetEnvAttr() is not supported in ODBC.
SQL_ATTR_PARAMOPT_ATOMIC	If specified, DB2 ODBC returns HYC00 on SQLSetConnectAttr() and HY011 on SQLGetConnectAttr().
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: The DB2 ODBC 3.0 driver does not support SQL_ONEPHASE. • **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 among the databases that support this protocol. Multiple readers and multiple updaters are allowed within a transaction. This attribute is only utilized when SQL_ATTR_CONNECTTYPE attribute is SQL_COORDINATED_TRANS. Then SQL_TWOPHASE is the default. This attribute is ignored when SQL_ATTR_CONNECTTYPE is set to SQL_CONCURRENT_TRANS. See *DB2 SQL Reference* for more information about distributed unit of work transactions. This attribute must be set before a connect request. Otherwise the attribute set request is 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. **Recommendation:** Insure that your application sets the SQL_ATTR_CONNECTTYPE attribute at the environment level rather than at a connection level.

Table 187. Connection attributes (continued)

Attribute	*ValuePtr*
SQL_ATTR_TXN_ISOLATION[4]	A 32-bit bit mask that sets the transaction isolation level for the current connection referenced by *hdbc*. The valid values for *vParam* can be determined at run time by calling SQLGetInfo() with *InfoType* set to SQL_TXN_ISOLATION_OPTION. The following values are accepted by DB2 ODBC, but each server might 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 UDB for iSeries. 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, see *IBM SQL Reference*. This attribute cannot be specified while there is an open cursor on any statement handle, or an outstanding transaction for this connection; otherwise, SQL_ERROR is returned on the function call (SQLSTATE **HY**011). For more information about setting this attribute, see “Setting isolation levels” on page 472. **Tip:** An IBM extension enables you to set transaction isolation levels on each individual statement handle. See the SQL_ATTR_STMTTXN_ISOLATION attribute in the function description for SQLSetStmtAttr().

Notes:

1. You can change the default value for this attribute with the AUTOCOMMIT keyword in the ODBC initialization file. See “AUTOCOMMIT” on page 52 for more information.
2. You can change the default value for this attribute with the CONNECTTYPE keyword in the ODBC initialization file. See “CONNECTTYPE” on page 53 for more information.
3. You can change the default value for this attribute with the MAXCONN keyword in the ODBC initialization file. See “MAXCONN” on page 56 for more information.
4. You can change the default value for this attribute with the TXNISOLATION keyword in the ODBC initialization file. See “TXNISOLATION” on page 61 for more information.

Return codes

After you call SQLSetConnectAttr(), it returns one of the following values:
- SQL_SUCCESS
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 188 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 188. SQLSetConnectAttr() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (SQLSetConnectAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
01S02	Option value changed.	SQL_ATTR_SYNC_POINT changed to SQL_TWOPHASE. SQL_ONEPHASE is not supported.
08S01	Unable to connect to data source.	The communication link between the application and the data source failed before the function completed.
08003	Connection is closed.	An *Attribute* value is specified that requires an open connection, but the *ConnectionHandle* is not in a connected state.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory for the specified handle.
HY009	Invalid use of a null pointer.	A null pointer is passed for *ValuePtr* and the value in **ValuePtr* is a string value.
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() is called with the statement handle, and returned SQL_NEED_DATA. This function is called before data is sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY011	Operation invalid at this time.	The argument *Attribute* is SQL_ATTR_TXN_ISOLATION and a transaction is open.
HY024	Invalid attribute value.	Given the specified *Attribute* value, an invalid value is specified in **ValuePtr*.
HY090	Invalid string or buffer length.	The *StringLength* argument is less than 0, but is not SQL_NTS.
HY092	Option type out of range.	The value specified for the argument *Attribute* is not valid for this version of DB2 ODBC.
HYC00	Driver not capable.	The value specified for the argument *Attribute* is a valid connection or statement attribute for this version of the DB2 ODBC driver, but is not supported by the data source.

Restrictions

None.

Example

The following example uses SQLConnectAttr() to set statement attribute values:

```
rc=SQLSetConnectAttr( hdbc,SQL_ATTR_AUTOCOMMIT,
                      (void*) SQL_AUTOCOMMIT_OFF, SQL_NTS);
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;
```

Related functions

The following functions relate to SQLSetConnectAttr() calls. Refer to the descriptions of these functions for more information about how you can use SQLSetConnectAttr() in your applications.

- “SQLAllocHandle() - Allocate a handle” on page 72
- “SQLGetConnectAttr() - Get current attribute setting” on page 196

- “SQLSetStmtAttr() - Set statement attributes” on page 367

SQLSetConnection() - Set connection handle

Purpose

Table 189. SQLSetConnection() specifications

ODBC	X/OPEN CLI	ISO CLI
No	No	No

This function is needed if the application needs to deterministically switch to a particular connection before continuing execution. Use this function only when your application mixes DB2 ODBC function calls with embedded SQL function calls and makes multiple database connections.

Syntax

```
SQLRETURN  SQLSetConnection  (SQLHDBC          hdbc);
```

Function arguments

Table 190 lists the data type, use, and description for each argument in this function.

Table 190. SQLSetConnection() arguments

Data type	Argument	Use	Description
SQLHDBC	*hdbc*	input	The connection handle associated with the connection to which the application wishes to switch.

Usage

ODBC allows multiple concurrent connections. It is not clear which connection an embedded SQL routine uses when invoked. In practice, the embedded routine uses the connection associated with the most recent network activity. However, from the application's perspective, this is not always easy to determine 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 ODBC 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 ODBC function applies.

Important: To mix DB2 ODBC with embedded SQL, you must not enable DB2 ODBC support for multiple contexts. The initialization file for mixed applications must specify MULTICONTEXT=0 or exclude MULTICONTEXT keyword.

For more information on using embedded SQL within DB2 ODBC applications see "Mixing embedded SQL with DB2 ODBC" on page 463.

Return codes

After you call SQLSetConnection(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR

- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 191 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 191. 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 is no specific SQLSTATE and for which the implementation does not define an SQLSTATE. SQLGetDiagRec() returns an error message in the argument *szErrorMsg* that describes the error and its cause.

Restrictions

None.

Example

See Figure 44 on page 417.

Related functions

The following functions relate to SQLSetConnection() calls. Refer to the descriptions of these functions for more information about how you can use SQLSetConnection() in your applications.

- “SQLConnect() - Connect to a data source” on page 121
- “SQLDriverConnect() - Use a connection string to connect to a data source” on page 142

SQLSetConnectOption() - Set connection option

Purpose

Table 192. SQLSetConnectOption() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	No

In the current version of DB2 ODBC, SQLSetConnectAttr() replaces SQLSetConnectOption(). See “SQLSetConnectAttr() - Set connection attributes” on page 346 for more information.

Although DB2 ODBC supports SQLSetConnectOption() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLSetConnectOption() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLSetConnectOption(
                              SQLHDBC          hdbc,
                              SQLUSMALLINT     fOption,
                              SQLUINTEGER      vParam);
```

Function arguments

Table 193 lists the data type, use, and description for each argument in this function.

Table 193. SQLSetConnectOption arguments

Data Type	Argument	Use	Description
HDBC	*hdbc*	input	Connection handle.
SQLUSMALLINT	*fOption*	input	Connect attribute to set.
SQLUINTEGER	*vParam*	input	Value associated with *fOption*. Depending on the attribute, this can be a 32-bit integer value, or a pointer to a nul-terminated string.

SQLSetCursorName() - Set cursor name

Purpose

Table 194. SQLSetCursorName() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	Yes

SQLSetCursorName() associates a cursor name with the statement handle. This function is optional because DB2 ODBC implicitly generates a cursor name when each statement handle is allocated.

Syntax

```
SQLRETURN   SQLSetCursorName (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCursor,
                              SQLSMALLINT       cbCursor);
```

Function arguments

Table 195 lists the data type, use, and description for each argument in this function.

Table 195. SQLSetCursorName() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLCHAR *	*szCursor*	input	Cursor name
SQLSMALLINT	*cbCursor*	input	The length, in bytes, of contents of *szCursor* argument

Usage

DB2 ODBC 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 ODBC maps this name to the internal name. The name remains associated with the statement handle, until the handle is dropped, or another SQLSetCursorName() is called on this statement handle.

Although SQLGetCursorName() returns the name set by the application (if one is set), error messages that are associated with positioned UPDATE and DELETE statements refer to the internal name.

Recommendation: Do not use SQLSetCursorName(). Instead, use the internal name, which you can obtain 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.)
- Because internally generated names begin with SQLCUR, SQL_CUR, or SQLCURQRS, the application must not input a cursor name starting with either SQLCUR or SQL_CUR in order to avoid conflicts with internal names.

- Because 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-Byte 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 are removed.

For efficient processing, applications should not include any leading or trailing spaces in the *szCursor* buffer. If the *szCursor* buffer contains a delimited identifier, applications should position the first double quote as the first character in the *szCursor* buffer.

Return codes

After you call SQLSetCursorName(), it returns one of the following values:
- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 196 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 196. SQLSetCursorName() SQLSTATEs

SQLSTATE	Description	Explanation
34000	Invalid cursor name.	This SQLSTATE is returned for one or more of the following reasons: • The cursor name specified by the argument *szCursor* is invalid. The cursor name either begins with SQLCUR, SQL_CUR, or SQLCURQRS 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 *szCursor* already exists. • The cursor name length is greater than the value returned by SQLGetInfo() with the SQL_MAX_CURSOR_NAME_LEN argument.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY009	Invalid use of a null pointer.	*szCursor* is a null pointer.
HY010	Function sequence error.	This SQLSTATE is returned for one or more of the following reasons: • There is an open or positioned cursor on the statement handle. • The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)

Table 196. SQLSetCursorName() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access the memory that is required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The argument *cbCursor* is less than **0**, but not equal to SQL_NTS.

Restrictions

None.

Example

Figure 31 shows an application that uses SQLSetCursorName() to set a cursor name.

```
/* ... */
    SQLCHAR         sqlstmt[] =
                    "SELECT name, job FROM staff "
                    "WHERE job='Clerk'  FOR UPDATE OF job";
/* ... */
    /* Allocate second statement handle for update statement */
    rc2 = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt2);

    /* Set Cursor for the SELECT statement's handle */
    rc = SQLSetCursorName(hstmt1, "JOBCURS", SQL_NTS);

    rc = SQLExecDirect(hstmt1, sqlstmt, SQL_NTS);

    /* bind name to first column in the result set */
    rc = SQLBindCol(hstmt1, 1, SQL_C_CHAR, (SQLPOINTER) name.s, 10,
                    &name.ind);

    /* bind job to second column in the result set */
    rc = SQLBindCol(hstmt1, 2, SQL_C_CHAR, (SQLPOINTER) job.s, 6,
                    &job.ind);

    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(newjob);
        if (newjob[0] != '\0') {
            sprintf(updstmt,
                    "UPDATE staff set job = '%s' where current of JOBCURS",
                    newjob);
            rc2 = SQLExecDirect(hstmt2, updstmt, SQL_NTS);
        }
    }
    if (rc != SQL_NO_DATA_FOUND)
        check_error(henv, hdbc, hstmt1, rc, __LINE__, __FILE__);
/* ... */
```

Figure 31. An application that sets a cursor name

Related functions

The following functions relate to SQLSetCursorName() calls. Refer to the descriptions of these functions for more information about how you can use SQLSetCursorName() in your applications.

- “SQLGetCursorName() - Get cursor name” on page 200

SQLSetEnvAttr() - Set environment attribute

Purpose

Table 197. SQLSetEnvAttr() specifications

ODBC	X/OPEN CLI	ISO CLI
No	Yes	Yes

SQLSetEnvAttr() sets attributes that govern aspects of environments.

Syntax

```
SQLRETURN  SQLSetEnvAttr (SQLHENV          EnvironmentHandle,
                          SQLINTEGER       Attribute,
                          SQLPOINTER       ValuePtr,
                          SQLINTEGER       StringLength);
```

Function arguments

Table 198 lists the data type, use, and description for each argument in this function.

Table 198. SQLSetEnvAttr() arguments

Data type	Argument	Use	Description
SQLHENV	*EnvironmentHandle*	input	Environment handle.
SQLINTEGER	*Attribute*	input	Environment attribute to set. See Table 199 on page 361 for the list of attributes and their descriptions.
SQLPOINTER	*ValuePtr*	input	The desired value for *Attribute*.
SQLINTEGER	*StringLength*	input	The length of *ValuePtr* in bytes if the attribute value is a character string. If *Attribute* does not denote a string, DB2 ODBC ignores *StringLength*.

Usage

When set, the attributes value affects all connections in this environment.

The application can obtain the current attribute value by calling SQLGetEnvAttr().

Table 199 on page 361 lists the SQLSetEnvAttr() *Attribute* values. The values that are shown in **bold** are default values.

Attribute values were renamed in ODBC 3.0. For a summary of the *Attributes* renamed in ODBC 3.0, see Table 257 on page 526.

Table 199. Environment attributes

Attribute	**Contents**
SQL_ATTR_ODBC_VERSION	A 32-bit integer that determines whether certain functionality exhibits ODBC 2.0 behavior or ODBC 3.0 behavior. This value cannot be changed while any connection handles are allocated. The following values are used to set the value of this attribute: • SQL_OV_ODBC3: Causes the following ODBC 3.0 behavior: – DB2 ODBC returns and expects ODBC 3.0 data type codes for date, time, and timestamp. – DB2 ODBC returns ODBC 3.0 SQLSTATE codes when SQLGetDiagRec() is called. – The *CatalogName* argument in a call to SQLTables() accepts a search pattern. • SQL_OV_ODBC2 causes the following ODBC **2.x** behavior: – DB2 ODBC returns and expects ODBC **2.x** data type codes for date, time, and timestamp. – DB2 ODBC returns ODBC 2.0 SQLSTATE codes when SQLGetDiagRec() or SQLError() 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 nul-termination in output arguments. The possible values are: • **SQL_TRUE**: DB2 ODBC uses nul-termination to indicate the length of output character strings. This is the default. • SQL_FALSE: DB2 ODBC does not use nul-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 no connection handles are allocated under the environment handle.

Table 199. Environment attributes (continued)

Attribute	**Contents**
SQL_ATTR_CONNECTTYPE[1]	A 32-bit integer value that specifies whether this application is to operate in a coordinated or uncoordinated distributed environment. The possible values are: • **SQL_CONCURRENT_TRANS**: 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 SQLEndTran() and not all of the connections commit successfully, the application is responsible for recovery. This corresponds to CONNECT (type 1) semantics subject to the restrictions described in "DB2 ODBC restrictions on the ODBC connection model" on page 11. This is the default. • SQL_COORDINATED_TRANS: The application wishes to have commit and rollbacks coordinated among multiple database connections. 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 is 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. **IBM specific:** This attribute is an IBM-defined extension. **Recommendation:** Have 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.
SQL_ATTR_MAXCONN[2]	A 32-bit integer value corresponding to the number that maximum concurrent connections that an application wants 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. The value that is in effect when the first connection is established is the value that is used. When the first connection is established, attempts to change this value are rejected. **IBM specific:** This attribute is an IBM-defined extension. **Recommendation:** Have 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 because this attribute is not supported in ODBC.

Notes:

1. You can change the default value for this attribute with the CONNECTTYPE keyword in the ODBC initialization file. See "CONNECTTYPE" on page 53 for more information.
2. You can change the default value for this attribute with the MAXCONN keyword in the ODBC initialization file. See "MAXCONN" on page 56 for more information.

Return codes

After you call SQLSetEnvAttr(), it returns one of the following values:
- SQL_SUCCESS
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 200 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 200. SQLSetEnvAttr() SQLSTATEs

SQLSTATE	Description	Explanation
HY009	Invalid use of a null pointer.	A null pointer is passed for *ValuePtr* and the value in **ValuePtr* is a string value.
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 is specified in **ValuePtr*.
HY090	Invalid string or buffer length.	The *StringLength* argument is less than 0, but is not SQL_NTS.
HY092	Option type out of range.	The value specified for the argument *Attribute* is not valid for this version of DB2 ODBC.
HYC00	Driver not capable.	The specified *Attribute* is not supported by DB2 ODBC. Given specified *Attribute* value, the value specified for the argument *ValuePtr* is not supported.

Restrictions

None.

Example

The following example uses SQLSetEnvAttr() to set an environment attribute. Also, see Figure 37 on page 401.

```
SQLINTEGER output_nts,autocommit;
rc = SQLSetEnvAttr( henv,
                   SQL_ATTR_OUTPUT_NTS,
                   ( SQLPOINTER ) output_nts,
                   0
                  ) ;
CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc );
```

Related functions

- “SQLAllocHandle() - Allocate a handle” on page 72
- “SQLGetEnvAttr() - Return current setting of an environment attribute” on page 224
- “SQLSetStmtAttr() - Set statement attributes” on page 367

SQLSetParam() - Bind a parameter marker to a buffer

Purpose

Table 201. SQLSetParam() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	No

In the current version of DB2 ODBC, SQLBindParameter() replaces SQLSetParam(). See "SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator" on page 85 for more information.

Although DB2 ODBC supports SQLSetParam() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLSetParam() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLSetParam      (SQLHSTMT          hstmt,
                               SQLUSMALLINT      ipar,
                               SQLSMALLINT       fCType,
                               SQLSMALLINT       fSqlType,
                               SQLUINTEGER       cbParamDef,
                               SQLSMALLINT       ibScale,
                               SQLPOINTER        rgbValue,
                               SQLINTEGER  FAR   *pcbValue);
```

Function arguments

Table 202 lists the data type, use, and description for each argument in this function.

Table 202. SQLSetParam() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLUSMALLINT	*ipar*	input	Parameter marker number, ordered sequentially left to right, starting at 1.

Table 202. SQLSetParam() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*fCType*	input	C data type of argument. 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_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_WCHAR Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type for the type indicated in *fSqlType*. See Table 4 on page 25 for more information.
SQLSMALLINT	*fSqlType*	input	SQL data type of column. The supported types are: • 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_ROWID • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC **Exceptions:** SQL_BLOB_LOCATOR, SQL_CLOB_LOCATOR, and SQL_DBCLOB_LOCATOR are application related concepts and do not map to a data type for column definition during a CREATE TABLE.

Table 202. SQLSetParam() arguments (continued)

Data type	Argument	Use	Description
SQLUINTEGER	*cbParamDef*	input	Precision of the corresponding parameter marker. If *fSqlType* denotes: • A binary or single-byte character string (for example, SQL_CHAR, SQL_BINARY), this is the maximum length in bytes for this parameter marker. • A double-byte character string (for example, SQL_GRAPHIC), this is the maximum length in double-byte characters for this parameter. • SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision. • SQL_ROWID, this must be set to 40, the maximum length in bytes for this data type. Otherwise, an error is returned. • Otherwise, this argument is ignored.
SQLSMALLINT	*ibScale*	input	Scale of the corresponding parameter marker if *fSqlType* is SQL_DECIMAL or SQL_NUMERIC. If *fSqlType* 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 *fSqlType* values mentioned here, *ibScale* is ignored.
SQLPOINTER	*rgbValue*	input (deferred)	Pointer to the location which contains (when the statement is executed) the actual values for the associated parameter marker.
SQLINTEGER *	*pcbValue*	input (deferred)	Pointer to the location which contains (when the statement is executed) the length, in bytes, of the parameter marker value stored at *rgbValue*. To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA. If *fCType* is SQL_C_CHAR, this storage location must contain either the exact length (in bytes) of the data stored at *rgbValue*, or SQL_NTS if the contents at *rgbValue* are nul-terminated. If *fCType* indicates character data (explicitly, or implicitly using SQL_C_DEFAULT), and this pointer is set to NULL, it is assumed that the application always provides a nul-terminated string in *rgbValue*. This also implies that this parameter marker never contains a null value. If *fSqlType* indicates a graphic data type, and the *fCType* is SQL_C_CHAR, the pointer to *pcbValue* can never be null and the contents of *pcbValue* can never hold SQL_NTS. In general for graphic data types, this length should be the number of bytes that the double-byte data occupies; therefore, the length must always be a multiple of 2. If this length is an odd number of bytes, then an error occurs when the statement is executed.

SQLSetStmtAttr() - Set statement attributes

Purpose

Table 203. SQLSetStmtAttr() specifications

ODBC	X/OPEN CLI	ISO CLI
3.0	Yes	Yes

SQLSetStmtAttr() sets attributes related to a statement. To set an attribute 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 204 lists the data type, use, and description for each argument in this function.

Table 204. SQLSetStmtAttr() arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLINTEGER	*Attribute*	input	Statement attribute to set. Refer to Table 205 on page 368 for a complete list of attributes.
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 nul-terminated character string. If the *Attribute* argument is a driver-specific value, the value in *ValuePtr* might be a signed integer.
SQLINTEGER	*StringLength*	input	Information about the **ValuePtr* argument. • For ODBC-defined attributes: – If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*. – If *ValuePtr* points to an integer, *BufferLength* is ignored. • For driver-defined attributes (IBM extension): – If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr* or SQL_NTS if it is a nul-terminated string. – If *ValuePtr* points to an integer, *BufferLength* is ignored.

Usage

Statement attributes for a statement remain in effect until they are changed by another call to SQLSetStmtAttr() or until the statement is dropped by calling SQLFreeHandle(). Calling SQLFreeStmt() with the SQL_CLOSE, SQL_UNBIND or the SQL_RESET_PARAMS attribute 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 ODBC returns SQL_SUCCESS_WITH_INFO and SQLSTATE **01**S02 (attribute value changed). To determine the substituted value, an application calls SQLGetStmtAttr().

The format of the information set with *ValuePtr* depends on the specified *Attribute*. SQLSetStmtAttr() accepts attribute information either in the format of a nul-terminated character string or a 32-bit integer value. The format of each *ValuePtr* value is noted in the attributes description shown in Table 205. This format applies to the information returned for each attribute in SQLGetStmtAttr(). Character strings that the *ValuePtr* argument of SQLSetStmtAttr() point to have a length of *StringLength*.

DB2 ODBC supports all of the ODBC 2.0 *Attribute* values that are renamed in ODBC 3.0. For a summary of the *Attribute* values renamed in ODBC 3.0, see Table 258 on page 526.

| **Overriding DB2 CCSIDs from DSNHDECP:** DB2 ODBC extensions to SQLSetStmtAttr() allow an application to override the Unicode, EBCDIC, or ASCII
| CCSID settings of the DB2 subsystem to which they are currently attached. This extension is intended for applications that are attempting to send and receive data to and from DB2 in a CCSID that differs from the default settings in the DB2 DSNHDECP.

The CCSID override applies only to input data bound to parameter markers through SQLBindParameter() and output data bound to columns through SQLBindCol().

The CCSID override applies on a statement level only. DB2 will continue to use the default CCSID settings in the DB2 DSNHECP after the statement is dropped or if SQL_CCSID_DEFAULT is specified.

You can use SQLGetStmtAttr() to query the settings of the current statement handle CCSID override.

Table 205 lists each *Attribute* value SQLSetStmtAttr() can set. Values shown in **bold** are default values.

Table 205. Statement attributes

Attribute	ValuePtr contents
SQL_ATTR_BIND_TYPE or SQL_ATTR_ROW_BIND_TYPE	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 (in bytes) of the structure or an instance of a buffer into which result columns are bound. For row-wise binding, the length (in bytes) 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 incriminated with the specified length, the result points 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.)

Table 205. Statement attributes (continued)

Attribute	ValuePtr contents
SQL_ATTR_CCSID_CHAR	A 32-bit integer value that specifies the Unicode, EBCDIC, or ASCII CCSID of input/output data, to or from a column of the following SQL data types: • SQL_CHAR • SQL_VARCHAR • SQL_LONGVARCHAR This CCSID will override the default CCSID setting from DB2 DSNHECP. The input data should be bound to parameter markers through SQLBindParameter(). The output data should be bound to columns through SQLBindCol(). SQL_CCSID_DEFAULT is the default value of this statement attribute, therefore the CCSIDs from the DB2 DSNHECP are used.
SQL_ATTR_CCSID_GRAPHIC	A 32-bit integer value that specifies the Unicode, EBCDIC, or ASCII CCSID of input/output data, to or from a column of the following SQL data types: • SQL_GRAPHIC • SQL_VARGRAPHIC • SQL_LONGVARGRAPHIC This CCSID overrides the default CCSID setting from DB2 DSNHECP. The input data should be bound to parameter markers through SQLBindParameter(). The output data should be bound to columns through SQLBindCol(). SQL_CCSID_DEFAULT is the default value of this statement attribute, therefore, the CCSIDs from the DB2 DSNHECP will be used. **IBM specific:** DB2 UDB for z/OS ODBC extensions to SQLSetStmtAttr() allow an application to override the CCSID settings of the DB2 subsystem to which they are currently attached. This extension is intended for applications that are attempting to send and receive data to and from DB2 in a CCSID that differs from the default settings in the DB2 DSNHDECP. The CCSID override applies only to input data bound to parameter markers through SQLBindParameter() and output data bound to columns through SQLBindCol(). The CCSID override applies on a statement level only. DB2 will continue to use the default CCSID settings in the DB2 DSNHECP after the statement is dropped or if SQL_CCSID_DEFAULT is specified. You can use SQLGetStmtAttr() to query the settings of the current statement handle CCSID override.
SQL_ATTR_CLOSE_BEHAVIOR	A 32-bit integer that forces the release of locks upon an underlying CLOSE CURSOR operation. The possible values are: • **SQL_CC_NO_RELEASE:** locks are not released when the cursor on this statement handle is closed. • 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 *fOption* argument set to SQL_CLOSE or SQLCloseCursor() is called. In addition, the end of the transaction (when a commit or rollback is issued) can also close the cursor (depending on the WITH HOLD attribute currently in use).

Table 205. Statement attributes (continued)

Attribute	ValuePtr contents
SQL_ATTR_CONCURRENCY	A 32-bit integer value that specifies the cursor concurrency: • **SQL_CONCUR_READ_ONLY** - Cursor is read-only. No updates are allowed. • SQL_CONCUR_LOCK - Cursor uses the lowest level of locking sufficient to ensure that the row can be updated. You cannot set this attribute if a cursor is open on the associated statement handle. **Unsupported attribute values:** ODBC architecture defines the following values, which are not supported by DB2 ODBC: • SQL_CONCUR_VALUES - Cursor uses optimistic concurrency control, comparing values. • SQL_CONCUR_ROWVER - Cursor uses optimistic concurrency control. If one of these values is used, SQL_SUCCESS_WITH_INFO (SQLSTATE **01**S02) is returned and the value remains unchanged.
SQL_ATTR_CURSOR_HOLD[1]	A 32-bit integer which specifies whether the cursor associated with this statement handle 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** • SQL_CURSOR_HOLD_OFF The default value when a statement handle is first allocated is SQL_CURSOR_HOLD_ON. This attribute cannot be specified while there is an open cursor on this statement handle. For more information about setting this attribute, see "Disabling cursor hold behavior" on page 472.
SQL_ATTR_CURSOR_TYPE	A 32-bit integer value that specifies the cursor type. The currently supported value is: • **SQL_CURSOR_FORWARD_ONLY** - Cursor behaves as a forward only scrolling cursor. This attribute cannot be set if there is an open cursor on the associated statement handle. **Unsupported attribute values:** ODBC architecture defines the following values, which are not supported by DB2 ODBC: • SQL_CURSOR_STATIC - The data in the result set appears to be static. • SQL_CURSOR_KEYSET_DRIVEN - The keys for the number of rows specified in the SQL_KEYSET_SIZE attribute is stored. DB2 ODBC does not support this attribute value. • SQL_CURSOR_DYNAMIC - The keys for the rows in the row set are saved. DB2 ODBC does not support this attribute value. If one of these values is used, SQL_SUCCESS_WITH_INFO (SQLSTATE **01**S02) is returned and the value remains unchanged.

Table 205. Statement attributes (continued)

Attribute	ValuePtr contents
SQL_ATTR_MAX_LENGTH	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_ATTR_MAX_LENGTH is less than the amount of data available, an SQLGetData() call or fetch returns 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 ODBC attempts to return all available data for character or binary type data.
SQL_ATTR_MAX_ROWS	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. For more information about setting this attribute, see “Result sets that are too large” on page 471.
SQL_ATTR_NODESCRIBE	A 32-bit integer which specifies whether DB2 ODBC should automatically describe the column attributes of the result set or wait to be informed by the application using SQLSetColAttributes(). • **SQL_NODESCRIBE_OFF** • SQL_NODESCRIBE_ON This attribute cannot be specified while there is an open cursor on this statement handle. This attribute is used in conjunction with the function SQLSetColAttributes() by an application which has prior knowledge of the exact nature of the result set to be returned and which does not wish to incur the extra network traffic associated with the descriptor information needed by DB2 ODBC to provide client side processing. **IBM specific:** This attribute is an IBM-defined extension.
SQL_ATTR_NOSCAN	A 32-bit integer value that specifies whether DB2 ODBC will scan SQL strings for escape clauses. The two permitted values are: • **SQL_NOSCAN_OFF** - SQL strings are scanned for escape clause sequences. • 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 eliminates some of the overhead processing associated with scanning.
SQL_ATTR_RETRIEVE_DATA	A 32-bit integer value indicating whether DB2 ODBC should actually retrieve data from the database when SQLExtendedFetch() is called. The possible values are: • **SQL_RD_ON**: SQLExtendedFetch() does retrieve data. • SQL_RD_OFF: SQLExtendedFetch() does not retrieve data. This is useful for verifying whether rows exist without incurring the overhead of sending long data from the database server. DB2 ODBC internally retrieves all the fixed-length columns, such as INTEGER and SMALLINT; so there is still some overhead. This attribute cannot be set if the cursor is open.
SQL_ATTR_ROW_ARRAY_SIZE	A 32-bit integer value that specifies the number of rows in the row set. This is the number of rows returned by each call to SQLExtendedFetch(). The default value is 1 which is equivalent to making a single SQLFetch() call. This attribute can be specified for an open cursor and becomes effective on the next SQLExtendedFetch() call.

Table 205. Statement attributes (continued)

Attribute	ValuePtr contents
SQL_ATTR_ROWSET_SIZE	A 32-bit integer value that specifies the number of rows in the row set. A row set 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 attribute can be specified even when the cursor is open and becomes effective on the next SQLExtendedFetch() call.
SQL_ATTR_STMTTXN_ISOLATION or SQL_ATTR_TXN_ISOLATION[2]	A 32-bit integer value that sets the transaction isolation level for the current statement handle. This overrides the default value set at the connection level (refer also to "SQLSetConnectOption() - Set connection option" on page 356 for the permitted values). This attribute cannot be set if there is an open cursor on this statement handle (SQLSTATE **24**000). **IBM specific:** The value SQL_ATTR_STMTTXN_ISOLATION is synonymous with SQL_ATTR_TXN_ISOLATION. SQL_ATTR_STMTTXN_ISOLATION is an IBM extension to allow setting this attribute at the statement level. For more information about setting this attribute, see "Setting isolation levels" on page 472.
| SQL_ATTR_CCSID_CHAR	A 32-bit integer value that specifies the Unicode, EBCDIC, or ASCII CCSID of input/output data, to or from a column of the following SQL data types: • SQL_CHAR • SQL_VARCHAR • SQL_LONGVARCHAR This CCSID will override the default CCSID setting from DB2 DSNHECP. The input data should be bound to parameter markers through SQLBindParameter(). The output data should be bound to columns through SQLBindCol(). SQL_CCSID_DEFAULT is the default value of this statement attribute, therefore the CCSIDs from the DB2 DSNHECP are used. |

Table 205. Statement attributes (continued)

Attribute	ValuePtr contents
| SQL_ATTR_CCSID_GRAPHIC	A 32-bit integer value that specifies the Unicode, EBCDIC, or ASCII CCSID of input/output data, to or from a column of the following SQL data types: • SQL_GRAPHIC • SQL_VARGRAPHIC • SQL_LONGVARGRAPHIC This CCSID overrides the default CCSID setting from DB2 DSNHECP. The input data should be bound to parameter markers through SQLBindParameter(). The output data should be bound to columns through SQLBindCol(). SQL_CCSID_DEFAULT is the default value of this statement attribute, therefore, the CCSIDs from the DB2 DSNHECP will be used. **IBM specific:** DB2 UDB for z/OS ODBC extensions to SQLSetStmtAttr() allow an application to override the CCSID settings of the DB2 subsystem to which they are currently attached. This extension is intended for applications that are attempting to send and receive data to and from DB2 in a CCSID that differs from the default settings in the DB2 DSNHDECP. The CCSID override applies only to input data bound to parameter markers through SQLBindParameter() and output data bound to columns through SQLBindCol(). The CCSID override applies on a statement level only. DB2 will continue to use the default CCSID settings in the DB2 DSNHECP after the statement is dropped or if SQL_CCSID_DEFAULT is specified. You can use SQLGetStmtAttr() to query the settings of the current statement handle CCSID override.

Notes:

1. You can change the default value for this attribute with the CURSORHOLD keyword in the ODBC initialization file. See “CURSORHOLD” on page 54 for more information.
2. You can change the default value for this attribute with the TXNISOLATION keyword in the ODBC initialization file. See “TXNISOLATION” on page 61 for more information.

Return codes

After you call SQLSetStmtAttr(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 206 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 206. SQLSetStmtAttr() SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (SQLSetStmtAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)

Table 206. SQLSetStmtAttr() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
01S02	Option value changed.	DB2 did not support the value specified in **ValuePtr*, or the value specified in **ValuePtr* is invalid due to SQL constraints or requirements. Therefore, DB2 ODBC substituted a similar value. (SQLSetStmtAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.)
08S01	Unable to connect to data source.	The communication link between the application and the data source failed before the function completed.
24000	Invalid cursor state.	The *Attribute* is SQL_ATTR_CONCURRENCY and the cursor is open.
HY000	General error.	An error occurred for which no specific SQLSTATE exists. The error message returned by SQLGetDiagRec() in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory for the specified handle.
HY009	Invalid use of a null pointer.	A null pointer is passed for *ValuePtr* and the value in **ValuePtr* is a string value.
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() is called with the statement handle, and returns SQL_NEED_DATA. This function is called before data is sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY011	Operation invalid at this time.	The *Attribute* is SQL_ATTR_CONCURRENCY and the statement is prepared.
HY024	Invalid attribute value.	Given the specified *Attribute* value, an invalid value is specified in **ValuePtr*.
HY090	Invalid string or buffer length.	The *StringLength* argument is less than 0, but is not SQL_NTS.
HY092	Option type out of range.	The value specified for the argument *Attribute* is not valid for this version of DB2 ODBC.
HYC00	Driver not capable.	The value specified for the argument *Attribute* is a valid connection or statement attribute for the version of the DB2 ODBC driver, but is not supported by the data source.

Restrictions

None.

Example

The following example uses SQLSetStmtAttr() to set statement attributes:

```
rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_CURSOR_HOLD,
                     ( void * ) SQL_CURSOR_HOLD_OFF,
                     0 ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

Related functions

The following functions relate to SQLSetStmtAttr() calls. Refer to the descriptions of these functions for more information about how you can use SQLSetStmtAttr() in your applications.

- “SQLCancel() - Cancel statement” on page 97
- “SQLGetConnectAttr() - Get current attribute setting” on page 196
- “SQLGetStmtAttr() - Get current setting of a statement attribute” on page 272
- “SQLSetConnectAttr() - Set connection attributes” on page 346

SQLSetStmtOption() - Set statement attribute

Purpose

Table 207. SQLSetStmtOption() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	No

In the current version of DB2 ODBC, SQLSetStmtAttr() replaces SQLSetStmtOption(). See “SQLSetStmtAttr() - Set statement attributes” on page 367 for more information.

Although DB2 ODBC supports SQLSetStmtAttr() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLSetStmtAttr() is available in the documentation for previous DB2 versions, which you can find at: www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLSetStmtOption (SQLHSTMT         hstmt,
                              SQLUSMALLINT     fOption,
                              SQLUINTEGER      vParam);
```

Function arguments

Table 208 lists the data type, use, and description for each argument in this function.

Table 208. SQLSetStmtOption() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLUSMALLINT	*fOption*	input	Attribute to set.
SQLUINTEGER	*vParam*	input	Value that is associated with *fOption*. *vParam* can be a 32-bit integer value or a pointer to a nul-terminated string.

SQLSpecialColumns() - Get special (row identifier) columns

Purpose

Table 209. SQLSpecialColumns() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	No

SQLSpecialColumns() returns unique row identifier information (primary key or unique index) for a table. The information is returned in an SQL result set, which you can retrieve by using the same functions that are used to process a result set that is generated by a query.

Syntax

```
SQLRETURN   SQLSpecialColumns(SQLHSTMT         hstmt,
                              SQLUSMALLINT     fColType,
                              SQLCHAR     FAR  *szCatalogName,
                              SQLSMALLINT      cbCatalogName,
                              SQLCHAR     FAR  *szSchemaName,
                              SQLSMALLINT      cbSchemaName,
                              SQLCHAR     FAR  *szTableName,
                              SQLSMALLINT      cbTableName,
                              SQLUSMALLINT     fScope,
                              SQLUSMALLINT     fNullable);
```

Function arguments

Table 210 lists the data type, use, and description for each argument in this function.

Table 210. SQLSpecialColumns() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLUSMALLINT	*fColType*	input	Type of unique row identifier to return. Only the following type is supported: • SQL_BEST_ROWID, which returns the optimal set of columns that can uniquely identify any row in the specified table. **Exception:** For compatibility with ODBC applications, SQL_ROWVER is also recognized, but not supported; therefore, if SQL_ROWVER is specified, an empty result is returned.
SQLCHAR *	*szCatalogName*	input	Catalog qualifier of a three-part table name. This must be a null pointer or a zero-length string.
SQLSMALLINT	*cbCatalogName*	input	The length, in bytes, of *szCatalogName*. This must be a set to 0.
SQLCHAR *	*szSchemaName*	input	Schema qualifier of the specified table.
SQLSMALLINT	*cbSchemaName*	input	The length, in bytes, of *szSchemaName*.
SQLCHAR *	*szTableName*	input	Table name.
SQLSMALLINT	*cbTableName*	input	The length, in bytes, of *cbTableName*.

Table 210. SQLSpecialColumns() arguments (continued)

Data type	Argument	Use	Description
SQLUSMALLINT	*fScope*	input	Minimum required duration for which the unique row identifier is valid. *fScope* 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 might not return a row if the row is updated or deleted by another transaction. • SQL_SCOPE_TRANSACTION: The row identifier is guaranteed to be valid for the duration of the current transaction. This attribute is only valid if SQL_TXN_SERIALIZABLE and SQL_TXN_REPEATABLE_READ isolation attributes are set. • SQL_SCOPE_SESSION: The row identifier is guaranteed to be valid for the duration of the connection. **Important:** This attribute is not supported by DB2 UDB for z/OS. 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, see *DB2 SQL Reference*.
SQLUSMALLINT	*fNullable*	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 can include columns where null values are permitted.

Usage

If multiple ways exist to uniquely identify any row in a table (that is, if the specified table is indexed with multiple unique indexes), DB2 ODBC returns the *best* set of row identifier column sets based on its internal criterion.

If no column set allows any row in the table to be uniquely identified, an empty result set is returned.

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. Table 211 on page 378 shows the order of the columns in the result set returned by SQLSpecialColumns(), sorted by SCOPE.

Because 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 are declared with a maximum length attribute of 128 bytes to be consistent with SQL92 limits. Because DB2 names are less than 128 bytes, the application can choose to always set aside 128 bytes (plus the nul-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 might be added and the names of the columns changed in future releases, the position of the current columns does not change. Table 211 lists these columns.

Table 211. Columns returned by SQLSpecialColumns()

Column number	Column name	Data type	Description
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 *fScope* argument: Actual scope of the row identifier. Contains one of the following values: • SQL_SCOPE_CURROW • SQL_SCOPE_TRANSACTION • SQL_SCOPE_SESSION See *fScope* in Table 210 on page 376 for a description of each value.
2	COLUMN_NAME	VARCHAR(128) NOT NULL	Name of the column that is (or part of) the table's primary key.
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 4 on page 25.
4	TYPE_NAME	VARCHAR(128) NOT NULL	DBMS character string represented of the name associated with DATA_TYPE column value.
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 Table 234 on page 509.
6	BUFFER_LENGTH	INTEGER	The maximum number of bytes for the associated C buffer to store data from this column if SQL_C_DEFAULT is specified on the SQLBindCol(), SQLGetData() and SQLBindParameter() calls. This length does not include any nul-terminator. For exact numeric data types, the length accounts for the decimal and the sign. See Table 236 on page 511.
7	DECIMAL_DIGITS	SMALLINT	The scale of the column. NULL is returned for data types where scale is not applicable. See Table 235 on page 510.

Table 211. Columns returned by SQLSpecialColumns() (continued)

Column number	Column name	Data type	Description
8	PSEUDO_COLUMN	SMALLINT	Indicates whether the column is a pseudo-column. DB2 ODBC only returns: • SQL_PC_NOT_PSEUDO DB2 DBMSs do not support pseudo columns. ODBC applications can receive the following values from other non-IBM RDBMS servers: • SQL_PC_UNKNOWN • SQL_PC_PSEUDO

Return codes

After you call SQLSpecialColumns(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 212 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 212. SQLSpecialColumns() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is opened on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length.	This SQLSTATE is returned for one or more of the following reasons: • The value of one of the length arguments is 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 *fColType* value is specified.
HY098	Scope type out of range.	An invalid *fScope* value is specified.
HY099	Nullable type out of range.	An invalid *fNullable* values is specified.
HYC00	Driver not capable.	DB2 ODBC does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

Figure 32 shows an application that prints a list of columns that uniquely define rows in a table. This application uses SQLSpecialColumns() to find these columns.

```
/* ... */
SQLRETURN
list_index_columns(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */
    rc = SQLSpecialColumns(hstmt, SQL_BEST_ROWID, NULL, 0, schema, SQL_NTS,
                    tablename, SQL_NTS, SQL_SCOPE_CURROW, SQL_NULLABLE);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) column_name.s, 129,
                    &column_name.ind);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) type_name.s, 129,
                    &type_name.ind);

    rc = SQLBindCol(hstmt, 5, SQL_C_LONG, (SQLPOINTER) & precision,
                    sizeof(precision), &precision_ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_SHORT, (SQLPOINTER) & scale,
                    sizeof(scale), &scale_ind);

    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");
        }
    }
/* ... */
```

Figure 32. An application that prints the column set for a unique index of a table

Related functions

The following functions relate to SQLSpecialColumns() calls. Refer to the descriptions of these functions for more information about how you can use SQLSpecialColumns() in your applications.

- “SQLColumns() - Get column information” on page 115
- “SQLStatistics() - Get index and statistics information for a base table” on page 381
- “SQLTables() - Get table information” on page 391

SQLStatistics() - Get index and statistics information for a base table

Purpose

Table 213. SQLStatistics() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	No

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 you can retrieve by using the same functions that you use to process a result set that is generated by a query.

Syntax

```
SQLRETURN   SQLStatistics    (SQLHSTMT          hstmt,
                               SQLCHAR     FAR   *szCatalogName,
                               SQLSMALLINT       cbCatalogName,
                               SQLCHAR     FAR   *szSchemaName,
                               SQLSMALLINT       cbSchemaName,
                               SQLCHAR     FAR   *szTableName,
                               SQLSMALLINT       cbTableName,
                               SQLUSMALLINT      fUnique,
                               SQLUSMALLINT      fAccuracy);
```

Function arguments

Table 214 lists the data type, use, and description for each argument in this function.

Table 214. SQLStatistics() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLCHAR *	*szCatalogName*	input	Catalog qualifier of a three-part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	*cbCatalogName*	input	The length, in bytes, of *cbCatalogName*. This must be set to 0.
SQLCHAR *	*szSchemaName*	input	Schema qualifier of the specified table.
SQLSMALLINT	*cbSchemaName*	input	The length, in bytes, of *szSchemaName*.
SQLCHAR *	*szTableName*	input	Table name.
SQLSMALLINT	*cbTableName*	input	The length, in bytes, of *cbTableName*.
SQLUSMALLINT	*fUnique*	input	Type of index information to return: • SQL_INDEX_UNIQUE Only unique indexes are returned. • SQL_INDEX_ALL All indexes are returned.

Table 214. SQLStatistics() arguments (continued)

Data type	Argument	Use	Description
SQLUSMALLINT	*fAccuracy*	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. Existing applications that specify this value receive the same results as SQL_QUICK. **Recommendation:** Do not use this value with new applications. • SQL_QUICK: Statistics which are readily available at the server are returned. The values might 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 statistics are 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 Table 215 in the order shown; the rows in the result set are ordered by NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME and ORDINAL_POSITION.

Because 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 are declared with a maximum length attribute of 128 bytes to be consistent with SQL92 limits. Because the length of DB2 names are less than 128 bytes, the application can choose to always set aside 128 bytes (plus the nul-terminator) for the output buffer. Alternatively, you can call SQLGetInfo() with the *InfoType* argument set to each of the following values:

- SQL_MAX_CATALOG_NAME_LEN, to determine the length of TABLE_CAT columns that the connected DBMS supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected DBMS supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected DBMS supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected DBMS supports

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change. Table 215 lists the columns in the result set SQLStatistics() currently returns.

Table 215. Columns returned by SQLStatistics()

Column number	Column name	Data type	Description
1	TABLE_CAT	VARCHAR(128)	The is always null.

Table 215. Columns returned by SQLStatistics() (continued)

Column number	Column name	Data type	Description
2	TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
3	TABLE_NAME	VARCHAR(128) NOT NULL	Name of the table.
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).
5	INDEX_QUALIFIER	VARCHAR(128)	The string is 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.
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.
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.
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.
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.
10	ASC_OR_DESC	CHAR(1)	Sort sequence for the column; A for ascending, D for descending. A null value is returned if the value in the TYPE column is SQL_TABLE_STAT.

Table 215. Columns returned by SQLStatistics() (continued)

Column number	Column name	Data type	Description
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.
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.
13	FILTER_CONDITION	VARCHAR(128)	If the index is a filtered index, this is the filter condition. Because DB2 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.

Important: 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 UDB for z/OS, the last time the RUNSTATS utility was run.)

Return codes

After you call SQLStatistics(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 216 on page 385 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 216. SQLStatistics() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is opened on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length.	This SQLSTATE is returned for one or more of the following reasons: • The value of one of the name length arguments is less than 0, but not equal to SQL_NTS. • The valid of one of the name length arguments exceeds the maximum value supported for that data source. You can obtain this maximum value with SQLGetInfo().
HY100	Uniqueness option type out of range.	An invalid *fUnique* value is specified.
HY101	Accuracy option type out of range.	An invalid *fAccuracy* value is specified.
HYC00	Driver not capable.	DB2 ODBC does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

Figure 33 on page 386 shows an application that prints the cardinality and the number of pages associated with a table. This application retrieves this information with SQLStatistics().

```
/* ... */
SQLRETURN
list_stats(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */
    rc = SQLStatistics(hstmt, NULL, 0, schema, SQL_NTS,
                tablename, SQL_NTS, SQL_INDEX_UNIQUE, SQL_QUICK);
    rc = SQLBindCol(hstmt, 4, SQL_C_SHORT,
                      &non_unique, 2, &non_unique_ind);
    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR,
                      index_name.s, 129, &index_name.ind);
    rc = SQLBindCol(hstmt, 7, SQL_C_SHORT,
                      &type, 2, &type_ind);
    rc = SQLBindCol(hstmt, 9, SQL_C_CHAR,
                      column_name.s, 129, &column_name.ind);
    rc = SQLBindCol(hstmt, 11, SQL_C_LONG,
                      &cardinality, 4, &card_ind);
    rc = SQLBindCol(hstmt, 12, SQL_C_LONG,
                      &pages, 4, &pages_ind);

    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");
    }
/* ... */
```

Figure 33. An application that prints page and cardinality information about a table

Related functions

The following functions relate to SQLStatistics() calls. Refer to the descriptions of these functions for more information about how you can use SQLStatistics() in your applications.

- "SQLColumns() - Get column information" on page 115
- "SQLSpecialColumns() - Get special (row identifier) columns" on page 376

SQLTablePrivileges() - Get table privileges

Purpose

Table 217. SQLTablePrivileges() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	No	No

SQLTablePrivileges() returns a list of tables and associated privileges for each table. The information is returned in an SQL result set, which you can retrieve by using the same functions that you use to process a result set that is generated by a query.

Syntax

```
SQLRETURN SQLTablePrivileges (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR     FAR   *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR     FAR   *szTableName,
                              SQLSMALLINT       cbTableName);
```

Function arguments

Table 218 lists the data type, use, and description for each argument in this function.

Table 218. SQLTablePrivileges() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLCHAR *	*szTableQualifier*	input	Catalog qualifier of a three-part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	*cbTableQualifier*	input	The length, in bytes, of *szCatalogName*. This must be set to 0.
SQLCHAR *	*szSchemaName*	input	Buffer that can contain a *pattern-value* to qualify the result set by schema name.
SQLSMALLINT	*cbSchemaName*	input	The length, in bytes, of *szSchemaName*.
SQLCHAR *	*szTableName*	input	Buffer that can contain a *pattern-value* to qualify the result set by table name.
SQLSMALLINT	*cbTableName*	input	The length, in bytes, of *szTableName*.

The *szSchemaName* and *szTableName* arguments accept search pattern. For more information about valid search patterns, see “Input arguments on catalog functions” on page 408.

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.

Because 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 are declared with a maximum length attribute of 128 bytes to be consistent with SQL92 limits. Because DB2 names are less than 128 bytes, the application can choose to always set aside 128 bytes (plus the nul-terminator) for the output buffer. Alternatively, you can call SQLGetInfo() with the *InfoType* argument set to each of the following values:

- SQL_MAX_CATALOG_NAME_LEN, to determine the length of TABLE_CAT columns that the connected DBMS supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected DBMS supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected DBMS supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected DBMS supports

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns remains unchanged. Table 219 lists the columns in the result set SQLTablePrivileges() currently returns.

Table 219. Columns returned by SQLTablePrivileges()

Column number	Column name	Data type	Description
1	TABLE_CAT	VARCHAR(128)	The is always null.
2	TABLE_SCHEM	VARCHAR(128)	The name of the schema contain TABLE_NAME.
3	TABLE_NAME	VARCHAR(128) NOT NULL	The name of the table.
4	GRANTOR	VARCHAR(128)	Authorization ID of the user who granted the privilege.
5	GRANTEE	VARCHAR(128)	Authorization ID of the user to whom the privilege is granted.
6	PRIVILEGE	VARCHAR(128)	The table privilege. This can be one of the following strings: • ALTER • CONTROL • DELETE • INDEX • INSERT • REFERENCES • SELECT • UPDATE
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.

The column names used by DB2 ODBC 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

After you call SQLTablePrivileges(), it returns one of the following values:
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see "Function return codes" on page 23.

Diagnostics

Table 220 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 220. SQLTablePrivileges() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is opened on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length.	This SQLSTATE is returned for one or more of the following reasons: • The value of one of the name length arguments is less than 0, but not equal to SQL_NTS. • The value 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 ODBC does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

Figure 34 on page 390 shows an application that uses SQLTablePrivileges() to generate a result set of privileges on tables.

```
/* ... */
SQLRETURN
list_table_privileges(SQLHDBC hdbc, SQLCHAR *schema,
                      SQLCHAR *tablename )
{
    SQLHSTMT        hstmt;
    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 */

    /* Allocate a statement handle to reference the result set */
    rc = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

    /* Create table privilges result set */
    rc = SQLTablePrivileges(hstmt, NULL, 0, schema, SQL_NTS,
                            tablename, SQL_NTS);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) grantor.s, 129,
                    &grantor.ind);

/* Continue Binding, then fetch and display result set */
/* ... */
```

Figure 34. An application that generates a result set containing privileges on tables

Related functions

The following functions relate to SQLTablePrivileges() calls. Refer to the descriptions of these functions for more information about how you can use SQLTablePrivileges() in your applications.

- “SQLTables() - Get table information” on page 391

SQLTables() - Get table information

Purpose

Table 221. SQLTables() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0	Yes	No

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          hstmt,
                              SQLCHAR     FAR   *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR     FAR   *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR     FAR   *szTableName,
                              SQLSMALLINT       cbTableName,
                              SQLCHAR     FAR   *szTableType,
                              SQLSMALLINT       cbTableType);
```

Function arguments

Table 222 lists the data type, use, and description for each argument in this function.

Table 222. SQLTables() arguments

Data type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLCHAR *	*szCatalogName*	input	Buffer that can contain a *pattern-value* to qualify the result set. *Catalog* is the first part of a three-part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	*cbCatalogName*	input	The length, in bytes, of *szCatalogName*. This must be set to 0.
SQLCHAR *	*szSchemaName*	input	Buffer that can contain a *pattern-value* to qualify the result set by schema name.
SQLSMALLINT	*cbSchemaName*	input	The length, in bytes, of *szSchemaName*.
SQLCHAR *	*szTableName*	input	Buffer that can contain a *pattern-value* to qualify the result set by table name.
SQLSMALLINT	*cbTableName*	input	The length, in bytes, of *szTableName*.

Table 222. SQLTables() arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*szTableType*	input	Buffer that can contain a *value list* to qualify the result set by table type. The value list is a list of uppercase comma-separated single quoted values for the table types of interest. Valid table type identifiers can include: TABLE, VIEW, SYSTEM TABLE, ALIAS, SYNONYM. If *szTableType* 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 any) are returned.
| SQLSMALLINT	*cbTableType*	input	Size of *szTableType*

Note that the *szCatalogName, szSchemaName*, and *szTableName* arguments accept search patterns. For more information about valid search patterns, see “Input arguments on catalog functions” on page 408.

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 *szSchemaName* argument can be applied: if *szSchemaName* is a string containing a single percent (%) character, and *szCatalogName* and *szTableName* are empty strings, then the result set contains a list of valid schemas in the data source.

If *szTableType* is a single percent character (%) and *szCatalogName, szSchemaName*, and *szTableName* 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 null values.)

If *szTableType* is not an empty string, it must contain a list of uppercase, comma-separated values for the types of interest; each value can be enclosed in single quotes or without single quotes. 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 *szSchemaName, szTableName*, and *szTableType* 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, or aliases, 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, SYSSCHEMA, TABLETYPE) can be specified in the DB2 ODBC initialization file to help restrict the result set when the application has supplied null pointers for either or both of *szSchemaName* and *szTableType*. These keywords and their usage are discussed in detail in “Initialization keywords” on page 51. If the application did not specify a null pointer for *szSchemaName* or *szTableType* then the associated keyword specification in the DB2 ODBC initialization file is ignored.

The result set returned by SQLTables() contains the columns listed in Table 223 in the order given. The rows are ordered by TABLE_TYPE, TABLE_CAT, TABLE_SCHEM, and TABLE_NAME.

Because 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 are declared with a maximum length attribute of 128 bytes to be consistent with SQL92 limits. BecauseDB2 names are less than 128 bytes, the application can choose to always set aside 128 bytes (plus the nul-terminator) for the output buffer. Alternatively, you can call SQLGetInfo() with the *InfoType* argument set to each of the following values:

- SQL_MAX_CATALOG_NAME_LEN, to determine the length of TABLE_CAT columns that the connected DBMS supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected DBMS supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected DBMS supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected DBMS supports

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns remains unchanged. Table 223 lists the columns in the result set SQLTables() currently returns.

Table 223. 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_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(762)	Contains the descriptive information about the table.

Return codes

After you call SQLTables(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

For a description of each of these return code values, see “Function return codes” on page 23.

Diagnostics

Table 224 on page 394 lists each SQLSTATE that this function generates, with a description and explanation for each value.

Table 224. SQLTables() SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is open on the statement handle.
40003 or **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate the required memory to support the execution or the completion of the function.
HY010	Function sequence error.	The function is called during a data-at-execute operation. (That is, the function is called during a procedure that uses the SQLParamData() or SQLPutData() functions.)
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to low internal resources.
HY090	Invalid string or buffer length.	This SQLSTATE is returned for one or more of the following reasons: • The value of one of the name length arguments is less than 0, but not equal to SQL_NTS. • The value of one of the name length arguments exceeds the maximum value supported for that data source. You can obtain this maximum value with SQLGetInfo().
HYC00	Driver not capable.	DB2 ODBC does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

Figure 35 on page 395 shows an application that uses SQLTables() to generate a result set of table name information that matches a search pattern. For another example, see “Querying environment and data source information” on page 36.

```
/* ... */
SQLRETURN init_tables(SQLHDBC hdbc )
{
    SQLHSTMT        hstmt;
    SQLRETURN       rc;

    SQLUSMALLINT     rowstat[MAX_TABLES];
    SQLUINTEGER      pcrow;

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

    /* SQL_ROWSET_SIZE sets the max number of result rows to fetch each time */
    rc = SQLSetStmtAttr(hstmt, SQL_ATTR_ROWSET_SIZE, (void*)MAX_TABLES, 0);

    /* Set size of one row, used for row-wise binding only */
    rc = SQLSetStmtAttr(hstmt, SQL_ATTR_BIND_TYPE,
                        (void *)sizeof(table) / MAX_TABLES, 0);

    printf("Enter Search Pattern for Table Schema Name:\n");
    gets(table->schem);
    printf("Enter Search Pattern for Table Name:\n");
    gets(table->name);
    rc = SQLTables(hstmt, NULL, 0, table->schem, SQL_NTS,
                   table->name, SQL_NTS, NULL, 0);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) &table->schem, 129,
                    &table->schem_l);

    rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) &table->name, 129,
                    &table->name_l);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) &table->type, 129,
                    &table->type_l);

    rc = SQLBindCol(hstmt, 5, SQL_C_CHAR, (SQLPOINTER) &table->remarks, 255,
                    &table->remarks_l);

    /* Now fetch the result set */
/* ... */
```

Figure 35. An application that returns a result set of table name information

Related functions

The following functions relate to SQLTables() calls. Refer to the descriptions of these functions for more information about how you can use SQLTables() in your applications.

- "SQLColumns() - Get column information" on page 115
- "SQLTablePrivileges() - Get table privileges" on page 387

SQLTransact() - Transaction management

Purpose

Table 225. SQLTransact() specifications

ODBC	X/OPEN CLI	ISO CLI
1.0 (Deprecated)	Yes	Yes

In the current version of DB2 ODBC, SQLEndTran() replaces SQLTransact(). See "SQLEndTran() - End transaction of a connection" on page 149 for more information.

Although DB2 ODBC supports SQLTransact() for backward compatibility, you should use current DB2 ODBC functions in your applications.

A complete description of SQLTransact() is available in the documentation for previous DB2 versions, which you can find at www.ibm.com/software/data/db2/zos/library.html.

Syntax

```
SQLRETURN   SQLTransact      (SQLHENV          henv,
                               SQLHDBC          hdbc,
                               SQLUSMALLINT     fType);
```

Function arguments

Table 226 lists the data type, use, and description for each argument in this function.

Table 226. SQLTransact() arguments

Data type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle. If *hdbc* is a valid connection handle, *henv* is ignored.
SQLHDBC	*hdbc*	input	Database connection handle. If *hdbc* is set to SQL_NULL_HDBC, then *henv* must contain the environment handle that the connection is associated with.
SQLUSMALLINT	*fType*	input	The desired action for the transaction. The value for this argument must be one of: • SQL_COMMIT • SQL_ROLLBACK

Chapter 5. Using advanced features

This chapter covers the following advanced tasks:

Setting and retrieving environment, connection, and statement attributes

Environments, connections, and statements each have a defined set of attributes (or options). You can query all these attributes, but you can change only some of these attributes from their default values. When you change attribute values, you change the behavior of DB2 ODBC.

The attributes that you can change are listed in the detailed descriptions of the set-attribute functions. For more information about these functions, see the following descriptions:

- "SQLSetEnvAttr() - Set environment attribute" on page 360
- "SQLSetConnectAttr() - Set connection attributes" on page 346
- "SQLSetStmtAttr() - Set statement attributes" on page 367
- "SQLSetColAttributes() - Set column attributes" on page 341

Read-only attributes (if any exist) are listed with the detailed function descriptions of the get-attribute functions. For information about some commonly used attributes, see "Programming hints and tips" on page 470.

Usually you write applications that use default attribute settings; however, these defaults are not always suitable for particular users of your application. DB2 ODBC provides two points at which users of your application can change default values of attributes at run time. Users specify attribute values either from an interface that uses the SQLDriverConnect() connection string or they can specify values in the DB2 ODBC initialization file.

The DB2 ODBC initialization file specifies the default attribute values for all DB2 ODBC applications. If an application does not provide users with an interface to the SQLDriverConnect() connection string, users can change default attribute values through the initialization file only. Attribute values that are specified with SQLDriverConnect() override the values that are set in the DB2 ODBC initialization file for any particular connection. For information about the connection string, see "SQLDriverConnect() - Use a connection string to connect to a data source" on page 142. For information about how the users of your applications can use the DB2 ODBC initialization file, as well as for a list of changeable defaults, see “DB2 ODBC initialization file” on page 49.

Important: The initialization file and connection string are intended for user tuning. Application developers should use the appropriate set-attribute functions to change attribute values. When you use set-attribute functions to set attribute values, the value that you specify overrides the initialization file value and the SQLDriverConnect() connection string value for that attribute.

Figure 36 shows how you set and retrieve attribute values within a basic connect scenario.

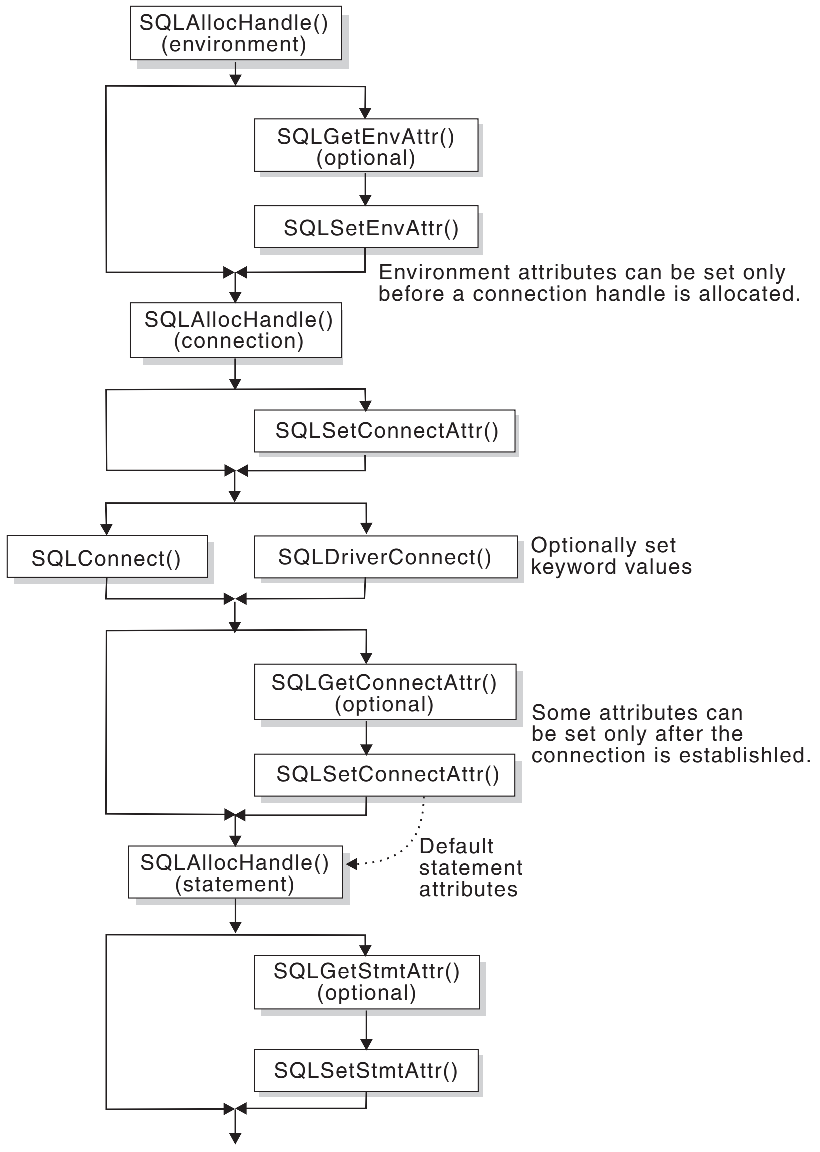

SQLAllocHandle() (environment)
SQLGetEnvAttr() (optional)
SQLSetEnvAttr()
Environment attributes can be set only before a connection handle is allocated.
SQLAllocHandle() (connection)
SQLSetConnectAttr()
SQLConnect()
SQLDriverConnect()
Optionally set keyword values
SQLGetConnectAttr() (optional)
SQLSetConnectAttr()
Some attributes can be set only after the connection is established.
SQLAllocHandle() (statement)
Default statement attributes
SQLGetStmtAttr() (optional)
SQLSetStmtAttr()

Figure 36. Setting and retrieving attributes

Setting and retrieving environment attributes

Attributes on an environment handle affect the behavior of all DB2 ODBC functions within that environment. You must set environment attributes before you allocate a connection handle. Because DB2 ODBC allows you to allocate only one environment handle, environment attributes affect all DB2 ODBC functions that your application calls.

To specify a new value for an environment attribute, call SQLSetEnvAttr().

To obtain the current value of an environment attribute, call SQLGetEnvAttr().

Setting and retrieving connection attributes

Attributes on a connection handle affect the behavior of all DB2 ODBC functions for that connection.

To change the value of a connection attribute, call SQLSetConnectAttr(). You can set a connection attribute only within one of the following periods of time. This period differs for each specific connection attribute.

- Any time after the connection handle is allocated
- Only before the actual connection is established
- Only after the connection is established
- After the connection is established only if that connection has no outstanding transactions or open cursors

For details about when you can set a specific connection attribute, see "SQLSetConnectAttr() - Set connection attributes" on page 346.

To obtain the current value of a connection attribute, call SQLGetConnectAttr().

Setting and retrieving statement attributes

Attributes on a statement handle affect the behavior of ODBC functions for that statement.

To change the value of a statement attribute, call SQLSetStmtAttr(). You can set a statement attribute only after you have allocated a statement handle. Statement attributes are one of the following types:

- Attributes that you can set, but currently only to one specific value
- Attributes that you can set any time after the statement handle is allocated
- Attributes that you can set only if no cursor is open on the statement handle

For details about each specific statement attribute, see "SQLSetStmtAttr() - Set statement attributes" on page 367.

Although you can use the SQLSetConnectAttr() function to set ODBC 2.0 statement attributes, setting statement attributes at the connection level is **not** recommended.

SQLGetConnectAttr() retrieves only connection attribute values; to retrieve the current value of a statement attribute you must call SQLGetStmtAttr().

Using a distributed unit of work

The transaction scenario that appears in "Connecting to one or more data sources" on page 12, portrays an application that can interact with only one data source in a transaction and perform only one transaction at a given time.

With a distributed unit of work (which is also called a coordinated distributed transaction), your application can access multiple database servers from within the same coordinated transaction. This section describes how you can write DB2 ODBC applications to use a distributed unit of work.

The environment and connection attribute SQL_ATTR_CONNECTTYPE controls whether your application operates in a coordinated or uncoordinated distributed environment. To change the distributed environment in which your application operates, you set this attribute to one of the following values:

- SQL_CONCURRENT_TRANS

  With this attribute value, the distributed environment is uncoordinated. Your application uses the semantics for a single data source per transaction, as described in Chapter 2, "Writing a DB2 ODBC application," on page 9. This value permits multiple (logical) concurrent connections to different data sources. SQL_CONCURRENT_TRANS is the default value for the SQL_ATTR_CONNECTTYPE environment attribute.
- SQL_COORDINATED_TRANS

  With this attribute value, the distributed environment is coordinated. Your application uses semantics for multiple data sources per transaction, as this section describes.

To use distributed units of work in your application, call SQLSetEnvAttr() or SQLSetConnectAttr() with the attribute SQL_ATTR_CONNECTTYPE set to SQL_COORDINATED_TRANS. You must set this attribute before you make a connection request.

All connections within an application must use the same connection type. You can set the connection type by using SQLSetEnvAttr(), SQLSetConnectAttr(), or the CONNECTTYPE keyword in the DB2 ODBC initialization file. For more information about the CONNECTTYPE keyword see "CONNECTTYPE" on page 53.

Recommendation: Set this environment attribute as soon as you successfully allocate an environment handle.

Establishing a distributed unit of work connection

You establish distributed unit of work connections when you call SQLSetEnvAttr() or SQLSetConnectAttr() with SQL_ATTR_CONNECTTYPE set to SQL_COORDINATED_TRANS. Also, you cannot specify MULTICONTEXT=1 in the initialization file if you want to use coordinated distributed transactions. Users of your application can specify CONNECTTYPE=2 in the DB2 ODBC initialization file or in the SQLDriverConnect() connection string to enable coordinated transactions. For information about the CONNECTTYPE keyword, see "DB2 ODBC initialization file" on page 49.

You cannot mix concurrent connections with coordinated connections in your application. The connection type that you specify for the first connection determines the connection type of all subsequent connections. SQLSetEnvAttr() and SQLSetConnectAttr() return an error if your application attempts to change the connection type while any connection is active. After you establish a connection type, it persists until you free all connection handles and change the value of the CONNECTTYPE keyword or the SQL_ATTR_CONNECTTYPE attribute.

Figure 37 on page 401 shows an example of an application that sets SQL_ATTR_CONNECTTYPE to SQL_COORDINATED_TRANS and performs a coordinated transaction on two data sources within the distributed environment.

```
/* ... */
#define MAX_CONNECTIONS   2

int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server);

int
main()
{
    SQLHENV         henv;
    SQLHDBC         hdbc[MAX_CONNECTIONS];
    SQLRETURN       rc;

    char *          svr[MAX_CONNECTIONS] =
                    {
                      "KARACHI"   ,
                      "DAMASCUS"
                    }

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

    /* Before allocating any connection handles, set Environment wide
       Connect Attributes  */
    /* Set to CONNECT(type 2)*/
    rc = SQLSetEnvAttr(henv, SQL_CONNECTTYPE,
                       (SQLPOINTER) SQL_COORDINATED_TRANS, 0);
/* ... */
    /* Connect to first data source */
    /* Allocate a connection handle     */
    if (SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc[0]) != SQL_SUCCESS) {
        printf(">---ERROR while allocating a connection handle-----\n");
        return (SQL_ERROR);
    }

    /* Connect to first data source (Type-II) */

    DBconnect (henv,
               &hdbc[0],
               svr[0]);


    /* Allocate a second connection handle     */
    if (SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc[1]) != SQL_SUCCESS) {
        printf(">---ERROR while allocating a connection handle-----\n");
        return (SQL_ERROR);
    }
    /* Connect to second data source (Type-II) */

    DBconnect (henv,
               &hdbc[1],
               svr[1]);
```

Figure 37. An application that connects to two data sources for a coordinated transaction (Part 1 of 2)

```
    /*********   Start processing step  *************************/
    /* Allocate statement handle, execute statement, and so on  */
    /* Note that both connections participate in the disposition*/
    /* of the transaction. Note that a NULL connection handle   */
    /* is passed as all work is committed on all connections.   */
    /*********   End processing step  ***************************/

    (void)SQLEndTran(SQL_HANDLE_HENV, henv, SQL_COMMIT);

    /* Disconnect, free handles and exit */
}


/********************************************************************
**   Server is passed as a parameter. Note that USERID and PASSWORD**
**   are always NULL.                                              **
********************************************************************/

int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server)
{
    SQLRETURN       rc;
    SQLCHAR         buffer[255];
    SQLSMALLINT     outlen;

    /* Allocate a connection handle     */
    SQLAllocHandle(SQL_HANDLE_DBC, henv, hdbc);
    rc = SQLConnect(*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf(">--- Error while connecting to database: %s -------\n", server);
        return (SQL_ERROR);
    } else {
        printf(">Connected to %s\n", server);
        return (SQL_SUCCESS);
    }
}
/* ... */
```

Figure 37. An application that connects to two data sources for a coordinated transaction (Part 2 of 2)

Setting attributes that govern distributed unit-of-work semantics

In distributed units of work, commits and rollbacks among multiple data source connections are coordinated. To establish coordinated connections, set the SQL_ATTR_CONNECTTYPE attribute to SQL_COORDINATED_TRANS or set the CONNECTTYPE keyword to 2. Coordinated connections are equivalent to connections that are established as CONNECT (type 2) in IBM embedded SQL.

All the connections within an application must have the same connection type, so in distributed unit of work, you must establish all connections as coordinated. The default commit mode for coordinated connections is manual-commit mode. (For a discussion about autocommit mode, see "Commit or rollback" on page 20.)

Figure 38 on page 403 shows the logical flow of an application that executes statements on two SQL_CONCURRENT_TRANS connections ('A' and 'B') and indicates the scope of the transactions. (Figure 39 on page 405 shows the logical flow and transaction scope of an application that executes the same statements on two SQL_COORDINATED_TRANS connections.)

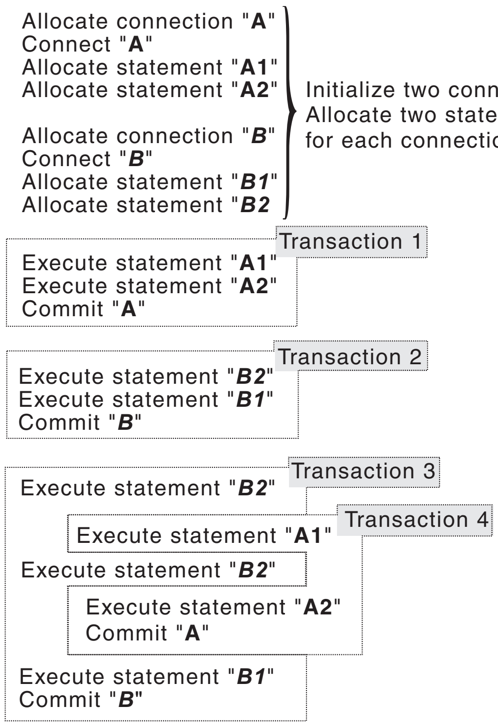


Figure 38. Multiple connections with concurrent transactions

In Figure 38, the third and fourth transactions are interleaved on multiple concurrent connections. If an application specifies SQL_CONCURRENT_TRANS, the ODBC model supports one transaction for each active connection. In Figure 38, the third transaction and the fourth transaction are managed and committed independently. (The third transaction consists of statements A1 and A2 at data source A and the fourth transaction consists of statements B2, B2 again, and B1 at data source B.) The transactions at A and B are independent and exist concurrently.

If you set the SQL_ATTR_CONNECTTYPE attribute to SQL_CONCURRENT_TRANS and specify MULTICONTEXT=0 in the initialization file, you can allocate any number of concurrent connection handles. However, only one physical connection to DB2 can exist at any given time with these settings. This behavior precludes support for the ODBC connection model. Consequently, applications that specify MULTICONTEXT=0 differ substantially from the ODBC execution model was previously described.

If an application specifies MULTICONTEXT=0 in the concurrent environment that Figure 38 portrays, the DB2 ODBC driver executes the third transaction as three separate implicit transactions. The DB2 ODBC driver performs these three implicit transactions with the following actions. (You do **not** issue these actions explicitly in your application).

- **First transaction**
  1. Executes statement B2
  2. Commits[1]
- **Second transaction**

1. Reconnects to data source B (after committing a transaction on data source A)
2. Executes statement B2
3. Commits[1]

- **Third transaction**
  1. Reconnects to data source B (after committing a transaction on data source A)
  2. Executes statement B1
  3. Commits[1]

Note:

1. In applications that run with MULTICONTEXT=0, you must always commit before changing data sources. You can specify AUTOCOMMIT=1 in the initialization file or call SQLSetConnectAttr() with SQL_ATTR_AUTOCOMMIT set to SQL_AUTOCOMMIT_ON to include these commit statements implicitly in your application. You can also explicitly include commits by using SQLEndTran() calls or SQL commit statements in your application.

From an application point of view, the transaction at data source B, which consists of statements B2, B2, and B1, becomes three independent transactions. The statements B2, B2, and B1 are each executed as independent transactions. Similarly, the fourth transaction at data source A, which consists of statements A1 and A2 becomes two independent transactions: A1 and A2.

For more information about multiple active transaction support, see “DB2 ODBC support of multiple contexts” on page 435.

Figure 39 on page 405 shows how the statements that Figure 38 on page 403 depicts are executed in a coordinated distributed environment. This figure shows statements on two SQL_COORDINATED_TRANS connections (’A’ and ’B’) and the scope of a coordinated distributed transaction.

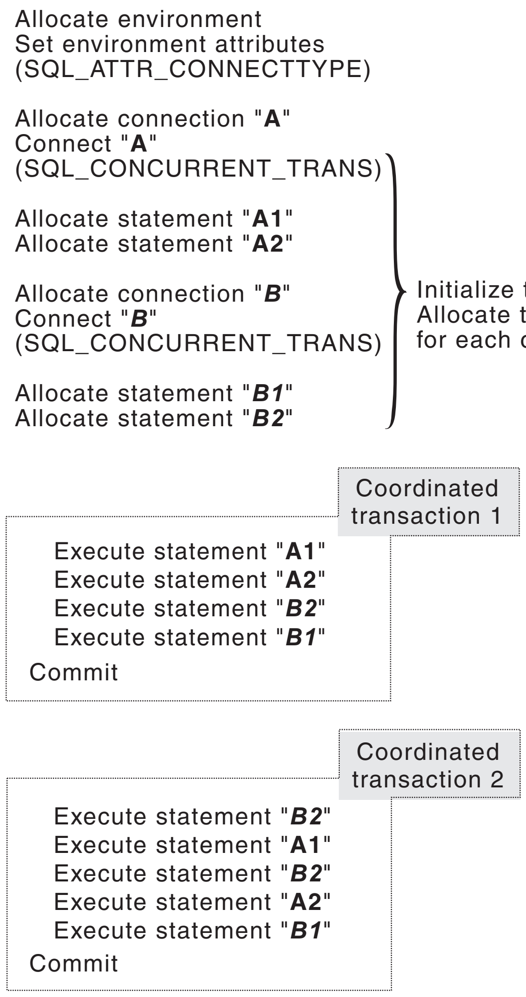


Figure 39. Multiple connections with coordinated transactions

Using global transactions

A global transaction is a recoverable unit of work, or transaction, that is made up of changes to a collection of resources. All resources that participate in a global transaction are guaranteed to be committed or rolled back as an atomic unit. z/OS Transaction Management and Resource Recovery Services (RRS) coordinates the updates that occur within a global transaction by using a two-phase commit protocol.

You include global transactions in your application to access multiple recoverable resources in the context of a single transaction. Global transactions enable you to write applications that participate in two-phase commit processing.

To enable global transactions, specify the keywords AUTOCOMMIT=0, MULTICONTEXT=0, and MVSATTACHTYPE=RRSAF in the initialization file.

To use global transactions, perform the following actions, which include RRS APIs, in your application:

1. Call ATRSENV() to provide environmental settings for RRS before you allocate connection handles.

2. Call ATRBEG() to mark the beginning of the global transaction.
3. Update the resources that are part of the global transaction.
4. Call SRRCMIT(), SRRBACK(), or the RRS service ATREND() to mark the end of the global transaction.
5. Repeat steps 2 and 4 for each global transaction that you include in your application.

SQLEndTran() is disabled within each global transaction, but you can still use this function to commit or rollback local transactions that are outside of the boundaries of the global transactions.

DB2 ODBC does not support global transaction processing for applications that run under a stored procedure.

For a complete description of RRS callable services, see *z/OS MVS Programming: Assembler Services Guide* or *z/OS MVS Programming: Resource Recovery*.

Figure 40 shows an application that uses global transaction processing. This application uses both ODBC and RRS APIs to make global transactions on two resources.

```
/* Provide environmental settings for RRS       */
ATRSENV();
/* Get an environment handle (henv)             */
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
/* Get a connection handle (hdbc1)              */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc1);
/* Get a connection handle (hdbc2)              */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc2);

/* Start a global transaction                   */
ATRBEG( ... , ATR_GLOBAL_MODE , ... );
/* Connect to STLEC1                            */
SQLConnect( hdbc1, "STLEC1", ... );

/* Execute some SQL with hdbc1                  */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
SQLExecDirect( hstmt1, ... );
SQLExecDirect( hstmt1, ... );
  .
  .

/* Connect to STLEC1B                           */
SQLConnect( hdbc2, "STLEC1B", ... );
/* Execute some SQL with hdbc2                  */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc2, &hstmt2);
SQLExecDirect( hstmt2, ... );
SQLExecDirect( hstmt2, ... );
   .
   .
```

Figure 40. An application that performs ODBC global transactions (Part 1 of 2)

```
/*  Free statement handles                      */
SQLFreeHandle(SQL_HANDLE_STMT, hstmt1);
SQLFreeHandle(SQL_HANDLE_STMT, hstmt2);

/*  Commit global transaction                   */
SRRCMIT();

/*  Start a global transaction                  */
ATRBEG( ... , ATR_GLOBAL_MODE , ... );
/*  Execute some SQL with hdbc1                 */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
SQLExecDirect( hstmt1, ... );
SQLExecDirect( hstmt1, ... );
   .
   .
/*  Execute some SQL with hdbc2                 */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc2, &hstmt2);
SQLExecDirect( hstmt2, ... );
SQLExecDirect( hstmt2, ... );
   .
   .
/*  Commit global transaction                   */
ATREND( ATR_COMMIT_ACTION );

/*  Disconnect hdbc1 and hdbc2                  */
SQLDisconnect( hdbc1 );
SQLDisconnect( hdbc2 );
```

Figure 40. An application that performs ODBC global transactions (Part 2 of 2)

Querying catalog information

Often, an application must obtain information from the catalog of the database management system. For example, many applications use catalog information to display a list of current tables for users to choose and manipulate. Although you can write your application to obtain this information with direct queries to the database management catalog, this approach is not advised.

Recommendation: Use DB2 ODBC catalog query functions and direct catalog queries to the DB2 ODBC shadow catalog to obtain catalog information.

When you use catalog query functions in your application, queries to the catalog are independent of the way that any single DBMS implements catalogs. As a result of this independence, applications that use these functions are more portable and require less maintenance.

You can also direct catalog query functions to the DB2 ODBC shadow catalog for improved performance.

The following sections describe the methods that you use to efficiently query the catalog:
- "Using the catalog query functions" on page 408
- "Directing catalog queries to the DB2 ODBC shadow catalog" on page 410

Using the catalog query functions

Catalog functions provide a generic interface to issue queries and return consistent result sets across the DB2 UDB family of servers. In most cases, this consistency allows you to avoid server-specific and release-specific catalog queries in your applications.

A catalog function is conceptually equivalent to an SQLExecDirect() function that executes a SELECT statement against a catalog table. Catalog functions return standard result sets through the statement handle on which you call them. Use SQLFetch() to retrieve individual rows from this result set as you would with any standard result set.

The following sections describe each DB2 ODBC function that queries the catalog and the result set that the function returns:

- "SQLColumnPrivileges() - Get column privileges" on page 110
- "SQLColumns() - Get column information" on page 115
- "SQLForeignKeys() - Get a list of foreign key columns" on page 178
- "SQLPrimaryKeys() - Get primary key columns of a table" on page 314
- "SQLProcedureColumns() - Get procedure input/output parameter information" on page 320
- "SQLProcedures() - Get a list of procedure names" on page 331
- "SQLSpecialColumns() - Get special (row identifier) columns" on page 376
- "SQLStatistics() - Get index and statistics information for a base table" on page 381
- "SQLTablePrivileges() - Get table privileges" on page 387
- "SQLTables() - Get table information" on page 391

Each of these functions return a result set with columns that are positioned in a specific order. Unlike column names, which can change as X/Open and ISO standards evolve, the position of these columns is static among ODBC drivers. When, in future releases, columns are added to these result sets, they will be added at the end position.

To make your application more portable, refer to columns by position when you handle result sets that catalog functions generate. Also, write your applications in such a way that additional columns do not adversely affect your application.

The CURRENTAPPENSCH keyword in the DB2 ODBC initialization file determines the encoding scheme for character data from catalog queries, as it does with all other result sets. See "CURRENTAPPENSCH" on page 53 and "Handling application encoding schemes" on page 443 for more details.

Some catalog functions execute fairly complex queries. For this reason, call these functions only when you need catalog information. Saving this information is better than making repeated queries to the catalog.

Input arguments on catalog functions

All of the catalog functions include the input arguments *CatalogName* and *SchemaName* (and their associated lengths). Catalog functions can also include the input arguments *TableName*, *ProcedureName*, and *ColumnName* (and their associated lengths). These input arguments either identify or constrain the amount of information that a catalog function returns. *CatalogName*, however, must always be a null pointer (with its length set to 0) because DB2 ODBC does not support three-part naming.

In the ″Function arguments″ section for these catalog functions, which appear in Chapter 4, "Functions," on page 63, each input argument is described either as a pattern-value argument or an ordinary argument. Argument descriptions vary between catalog functions. For example, SQLColumnPrivileges() treats *SchemaName* and *TableName* as ordinary arguments, whereas SQLTables() treats these arguments as pattern-value arguments.

Ordinary arguments are inputs that are taken literally. These arguments are case-sensitive. Ordinary arguments do not qualify a query, but rather they explicitly identify the input information. If you pass a null pointer to this type of argument, the results are unpredictable.

Pattern-value arguments constrain the size of the result set as though the underlying query were qualified by a WHERE clause. If you pass a null pointer to a pattern-value input, that argument is not used to restrict the result set (that is, no WHERE clause restricts the query). If a catalog function has more than one pattern-value input argument, these arguments are treated as though the WHERE clauses in the underlying query were joined by AND. A row appears in the result set only if it meets the conditions of all pattern-value arguments that the catalog function specifies.

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.
- Characters that stand for themselves. The case of a letter is significant.

These argument values are used on conceptual LIKE predicates in the WHERE clause. To treat metadata characters (_ and %) literally, you must include an escape character immediately before the _ or % character. To use the escape character itself as a literal part of a pattern-value argument, include the escape character twice in succession. You can determine the escape character that an ODBC driver uses by calling SQLGetInfo() with the *InfoType* argument set to SQL_SEARCH_PATTERN_ESCAPE.

| You can use catalog functions with EBCDIC, Unicode, and ASCII encoding
| schemes. The CURRENTAPPENSCH keyword in the initialization file determines
| which one of these encoding schemes you use. For EBCDIC, Unicode UTF-8, and
| ASCII input strings use generic catalog functions. For UCS-2 input strings, use
| suffix-W catalog functions. For each generic catalog function, a corresponding
| suffix-W API provides UCS-2 support. For more information, see "Choosing an API
| entry point" on page 444.

Catalog functions example

The sample output in Figure 41 on page 410 shows the following information:
- A list of all tables for the specified schema (qualifier) name or search pattern
- Column, special column, foreign key, and statistics information for a selected table

In Chapter 4, "Functions," on page 63, each catalog function description includes a relevant section of the code that generated this output.

```
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<<
```

Figure 41. Sample output from an application that uses catalog functions

Directing catalog queries to the DB2 ODBC shadow catalog

| You use the DB2 ODBC shadow catalog for increased performance when you need
| information from the catalog. To increase the performance of an application that
| must frequently query the catalog, implement the DB2 ODBC shadow catalog and
| redirect catalog functions to the shadow catalog instead of using the native DB2
| catalog.

The shadow catalog consists of a set of pseudo-catalog tables that contain rows that represent objects that are defined in the DB2 catalog. These tables are pre-joined and indexed to provide faster catalog access for ODBC applications. Tables in the shadow catalog contain only the columns that are necessary for supporting ODBC operations.

CLISCHEM is the default schema name for tables that make up the DB2 ODBC shadow catalog. To redirect catalog functions to access these base DB2 ODBC shadow catalog tables, add the entry CLISCHEMA=CLISCHEM to the data source section of the DB2 ODBC initialization file as follows:

```
[DATASOURCE]
MVSDEFAULTSSID=V61A
CLISCHEMA=CLISCHEM
```

Optionally, you can create views for the DB2 ODBC shadow catalog tables that are qualified with your own schema name, and redirect the ODBC catalog functions to access these views instead of the base DB2 ODBC shadow catalog tables. To redirect the catalog functions to access your own set of views, add the entry CLISCHEMA=*myschema* (where *myschema* is the schema name of the set of views that you create) to the data source section of the DB2 ODBC initialization file as follows:

```
[DATASOURCE]
MVSDEFAULTSSID=V61A
CLISCHEMA=PAYROLL
APPLTRACE=1
APPLTRACEFILENAME="DD:APPLTRC"
```

You can use the CREATE VIEW SQL statement to create views of the DB2 ODBC shadow catalog tables. To use your own set of views, you must create a view for each DB2 ODBC shadow catalog table.

Example: Execute the following SQL statement to create a view, where *table_name* is the name of a DB2 ODBC shadow catalog table:

```
CREATE VIEW PAYROLL.table_name AS
  SELECT * FROM PAYROLL.table_name WHERE TABLE_SCHEM='USER01';
```

Table 227 lists the base DB2 ODBC shadow catalog tables and the catalog functions that access these tables.

Table 227. Shadow catalog tables and DB2 ODBC APIs

Shadow catalog table	DB2 ODBC catalog function
CLISCHEM.COLUMNPRIVILEGES	SQLColumnPrivileges()
CLISCHEM.COLUMNS	SQLColumns()
CLISCHEM.FOREIGNKEYS	SQLForeignKeys()
CLISCHEM.PRIMARYKEYS	SQLPrimaryKeys()
CLISCHEM.PROCEDURECOLUMNS	SQLProcedureColumns()
CLISCHEM.PROCEDURES	SQLProcedures()
CLISCHEM.SPECIALCOLUMNS	SQLSpecialColumns()
CLISCHEM.TSTATISTICS	SQLStatistics()
CLISCHEM.TABLEPRIVILEGES	SQLTablePrivileges()
CLISCHEM.TABLE	SQLTables()

Creating and maintaining the DB2 ODBC shadow catalog

IBM DB2 DataPropagator for z/OS populates and maintains the DB2 ODBC shadow catalog. DB2 UDB for z/OS supports the DATA CAPTURE CHANGE clause of the ALTER TABLE SQL statement for DB2 catalog tables. This support allows DB2 to mark log records that are associated with any statements that change the DB2 catalog (for example, CREATE and DROP). In addition, the DB2 DataPropagator Capture and Apply process identifies and propagates the DB2 catalog changes to the DB2 ODBC shadow, based on marked log records.

For detailed information about setting up the DB2 ODBC shadow catalog and running IBM DB2 DataPropagator for z/OS, see *DB2 Universal Database Replication Guide and Reference*.

Shadow catalog example

If you specify CLISCHEMA=PAYROLL in the data source section of the DB2 ODBC initialization file, the ODBC catalog functions that normally query the DB2 catalog tables (SYSIBM schema) reference the following set of views of the ODBC shadow catalog base tables:

- PAYROLL.COLUMNS
- PAYROLL.TABLES
- PAYROLL.COLUMNPRIVILEGES
- PAYROLL.TABLEPRIVILEGES
- PAYROLL.SPECIALCOLUMNS
- PAYROLL.PRIMARYKEYS
- PAYROLL.FOREIGNKEYS
- PAYROLL.TSTATISTICS
- PAYROLL.PROCEDURES
- PAYROLL.PROCEDURECOLUMNS

Sending or retrieving long data values in pieces

When an application must manipulate long data values, loading these entire values into storage can become unfeasible. For this reason, DB2 ODBC provides a technique that enables you to handle long data values in pieces. This technique, called *specifying parameter values at execute time*, is the same method that you can use to specify values for fixed-size non-character data types, such as integers.

Figure 42 on page 413 depicts both the processes of sending data in pieces and retrieving data in pieces. The right side of the figure shows the process that you use to send data in pieces; the left side of the figure shows the process that you use to retrieve data in pieces.

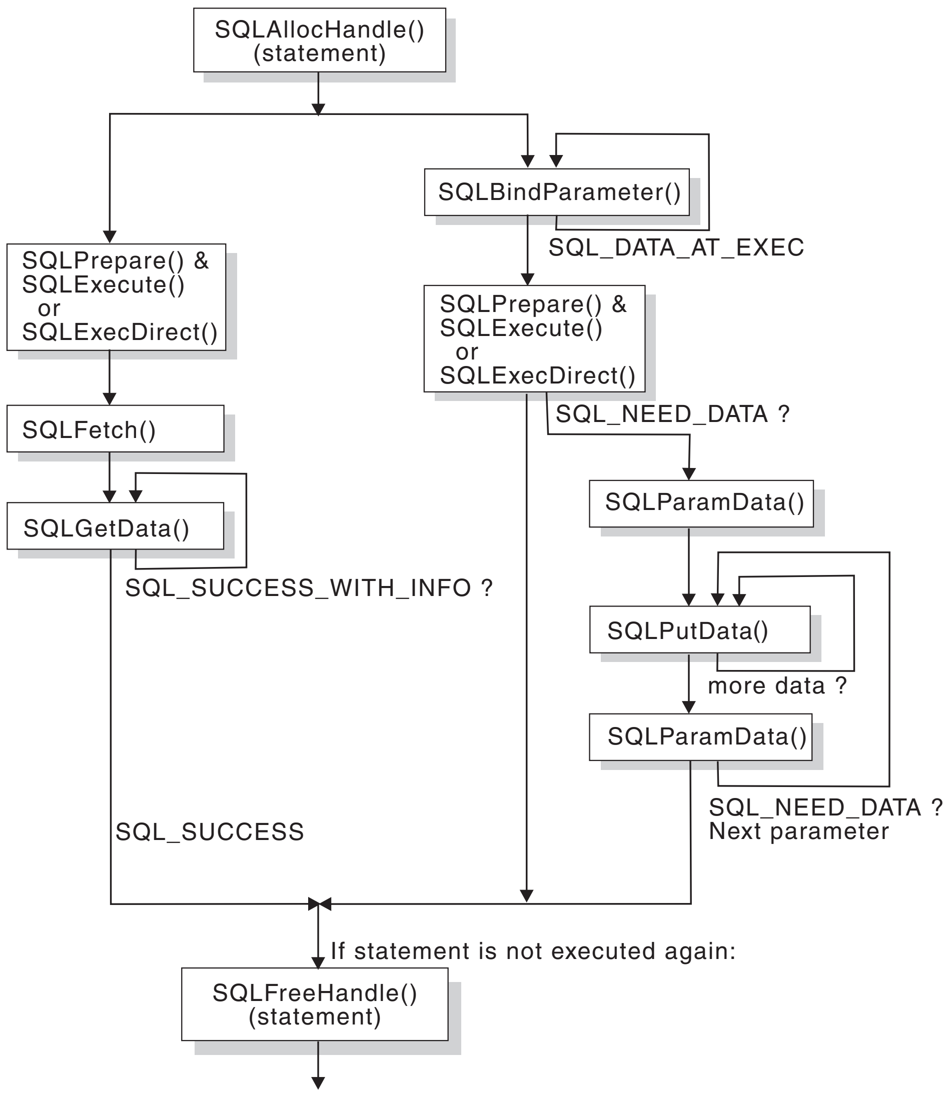

SQLAllocHandle() (statement)
SQLBindParameter()
SQL_DATA_AT_EXEC
SQLPrepare() & SQLExecute() or SQLExecDirect()
SQLFetch()
SQLGetData()
SQL_SUCCESS_WITH_INFO ?
SQLPrepare() & SQLExecute() or SQLExecDirect()
SQL_NEED_DATA ?
SQLParamData()
SQLPutData()
more data ?
SQLParamData()
SQL_NEED_DATA ? Next parameter
SQL_SUCCESS
If statement is not executed again:
SQLFreeHandle() (statement)

Figure 42. Input and retrieval of data in pieces

Specifying parameter values at execution time

A data-at-execute parameter is a bound parameter for which a value is prompted at execution time. Normally, you store a value in memory to use for a parameter before you call SQLExecute() or SQLExecDirect(). To create data-at-execute parameters, call SQLBindParameter() and specify both of the following arguments for each data-at-execute parameter you want to create:

- Set the *pcbValue* argument to SQL_DATA_AT_EXEC.
- Set the *rgbValue* argument to a value you can use to uniquely identify the parameter for which you need to supply data. This value names that parameter so that you can refer to it later.

SQLExecDirect() and SQLExecute() return SQL_NEED_DATA for statements that contain data-at-execute parameters to prompt you to supply a value. When SQLExecDirect() or SQLExecute() returns SQL_NEED_DATA, you must perform the following steps in your application:

1. Call SQLParamData() to conceptually advance to the first such parameter. SQLParamData() returns SQL_NEED_DATA and provides the value of the input *rgbValue* buffer that you specified in the SQLBindParameter() call. This value helps you identify the information that you need to supply.
2. Call SQLPutData() to pass the actual data for the parameter. You call SQLPutData() repeatedly to send long data in pieces.
3. Call SQLParamData() after you provide the entire data for this data-at-execute parameter. If additional data-at-execute parameters need data, SQLParamData() returns SQL_NEED_DATA. Repeat steps 2 and 3 until SQLParamData() returns SQL_SUCCESS.

When all data-at-execute parameters are assigned values, SQLParamData() completes execution of the SQL statement. SQLParamData() also produces a return value and diagnostics as the original SQLExecDirect() or SQLExecute() would have produced. The right side of Figure 42 on page 413 illustrates this flow.

While the data-at-execution flow is in progress, you can call only the following DB2 ODBC functions:

- SQLParamData() and SQLPutData(), as the previous procedure to specify parameter values at execute time describes.
- SQLCancel(), which is used to cancel the flow and force an exit from the loops on the right side of Figure 42 on page 413 without executing the SQL statement.
- SQLGetDiagRec()

You cannot terminate the transaction nor set connection attributes in a data-at-execution flow.

Fetching data in pieces

Typically to retrieve data, you allocate application variables to hold the data you retrieve, and you call SQLBindCol() to associate these application variables with a column in a result set. Based on the size of the values that a column contains, you choose the amount of memory that values from this column can occupy in your application. (To determine the size of the largest value in a specific result column, call SQLDescribeCol(). The output argument *pcbColDef* returns this information.)

In the case of character and binary data, columns can contain large values. If the size of a column value exceeds the size of the buffer that you allocate, you can call SQLGetData() repeatedly to obtain this value in a sequence of pieces that are more manageable in size.

As the left side Figure 42 on page 413 depicts, SQLGetData() returns SQL_SUCCESS_WITH_INFO (with SQLSTATE **01**004) to indicate that more data exists for this column. Call SQLGetData() repeatedly to retrieve the remaining pieces of data. When you retrieve the final piece of data, SQLGetData() returns SQL_SUCCESS.

Using arrays to pass parameter values

In data entry and update applications, users might often insert, delete, or alter many cells in a data entry form before they send these changes to the database. For these situations, DB2 ODBC provides an array input method that eliminates the need for you to call SQLExecute() repeatedly on the same INSERT, DELETE, or UPDATE statement. In addition, the use of arrays to pass parameter values can reduce network flows.

You pass arrays to parameter markers with the following method:

1. Call SQLBindParameter() for each parameter marker that you bind to an input array in memory. Use the following argument values in this function call:
   - Set the *fParamType* argument value to SQL_PARAM_INPUT.
   - Point the *rgbValue* argument to the array that contains input data for the parameter marker.
   - For character and binary input data, specify the length, in bytes, of each element in the input array with the input argument *cbValueMax*. (For other input data types, this argument is ignored.)
   - Optionally, point the *pcbValue* argument to an array that contains the lengths, in bytes, of each value in the input array. Specify each length value in the *pcbValue* array to be the length of the corresponding value in the *rgbValue* array.
2. Call SQLParamOptions() and specify, in the *crow* argument, the number of rows that the input array contains. This value indicates the number of different values for each parameter.
3. Call SQLExecute() to send all the parameter values to the database.

When you insert and update rows with arrays, use SQLRowCount() to verify the number of rows you changed.

Queries with parameter markers that are bound to arrays on the WHERE clause generate multiple sequential result sets. You process each result set that such a query returns individually. After you process the initial result set, call SQLMoreResults() to retrieve each additional result set. See “SQLMoreResults() - Check for more result sets” on page 289 for more information.

Example: Consider an application that performs an array insert, as the right side of Figure 43 on page 416 illustrates. Suppose that this application enables users 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. This application can then request to prepare the following SQL statement:

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

Because this statement contains three parameter markers, the application uses three arrays to store input data. When the user makes changes to *n* rows, the application places *n* values in each array. When the user decides to send these changes to the database, the application binds the parameter markers in the prepared SQL statement to the arrays. The application then calls SQLParamOptions() with the *crow* argument set to *n*. This value specifies the number of elements in each array.

Figure 43 on page 416 shows the two methods of executing a statement with *m* parameters *n* times. Both methods must call SQLBindParameter() once for each parameter.

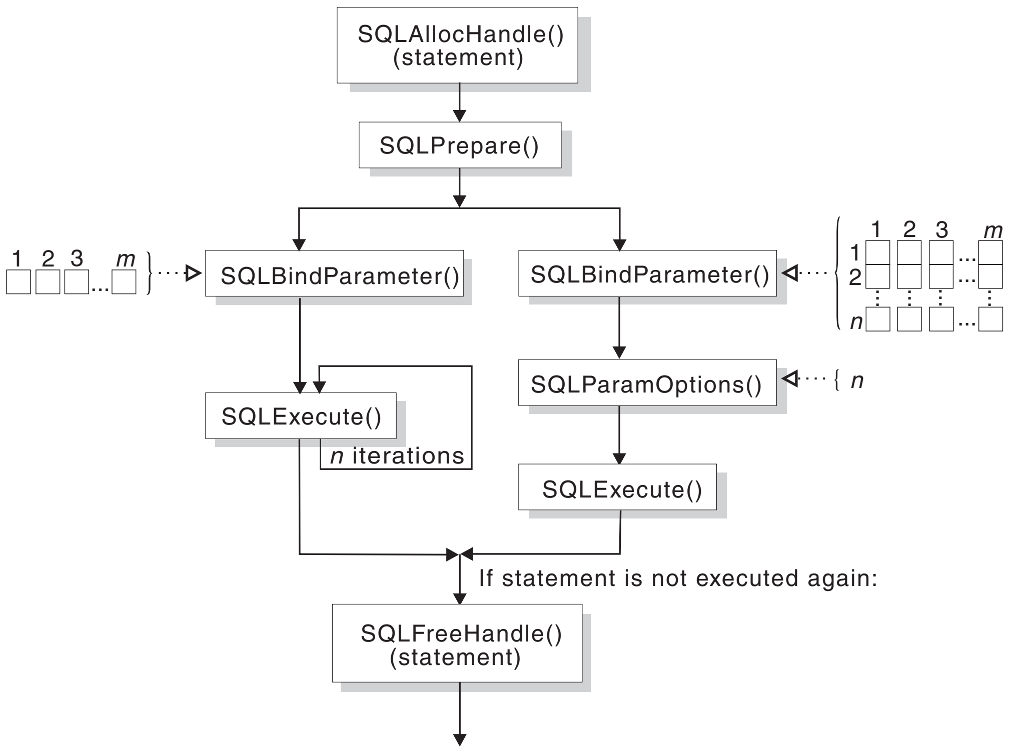


Figure 43. Array insert

The left side of Figure 43 illustrates a method of bulk operations that does not use arrays to pass parameter values. SQLBindParameter() binds each parameter marker to a host variable that contains a single value. Because this method does not perform array inserts, SQLExecute() is called repeatedly. Before each SQLExecute() call, the application updates the variables that are bound to the input parameters. This method calls SQLExecute() to execute every operation.

The right side of Figure 43 illustrates a method of bulk operations that uses arrays to pass parameter values. SQLExecute() is called only once for any number of bulk operations. The array method calls SQLParamOptions() to specify the number of rows (*n*), and then it calls SQLExecute().

Figure 44 on page 417 shows an array INSERT statement. For an example of an array SELECT statement, see “SQLMoreResults() - Check for more result sets” on page 289.

```
/* ... */
    SQLUINTEGER pirow = 0;
    SQLCHAR          stmt[] =
    "INSERT INTO CUSTOMER ( Cust_Num, First_Name, Last_Name) "
      "VALUES (?, ?, ?)";

    SQLINTEGER       Cust_Num[25] = {
        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[25][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"
    };
```

Figure 44. An application that performs an array insert (Part 1 of 2)

```
    SQLCHAR          Last_Name[25][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);

    rc = SQLParamOptions(hstmt, 25, &pirow);

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

    rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          31, 0, First_Name, 31, NULL);

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

    rc = SQLExecute(hstmt);
    printf("Inserted %ld Rows\n", pirow);

/* ... */
```

Figure 44. An application that performs an array insert (Part 2 of 2)

Retrieving a result set into an array

One of the most common tasks that an application performs is to issue a query statement and then fetch rows from the result set that the query generates. To fetch rows, you typically bind application variables to columns in the result set with SQLBindCol(). Then you individually fetch each row into these application variables. If you want to store more than one row from the result set in your application, you can follow each fetch with an additional operation. You can save previously fetched values in your application by using one of the following operations before you fetch additional data:

- Copy fetched values to application variables that are not bound to a result set
- Call a new set of SQLBindCol() functions to assign new application variables to the next fetch

If you do not use one of these operations, each fetch replaces the values that you previously retrieved.

Alternatively, you can retrieve multiple rows of data (called a row set) simultaneously into an array. This method eliminates the overhead of extra data copies or SQLBindCol() calls. SQLBindCol() can bind an array of application variables. By default, SQLBindCol() binds rows in column-wise fashion: this type of bind is similar to using SQLBindParameter() to bind arrays of input parameter values, as described in the previous section. You can also bind data in a row-wise fashion to retrieve data into an array.

Returning array data for column-wise bound data

Figure 45 is a logical view of column-wise binding. The right side of Figure 47 on page 420 shows the function flows for column-wise retrieval.

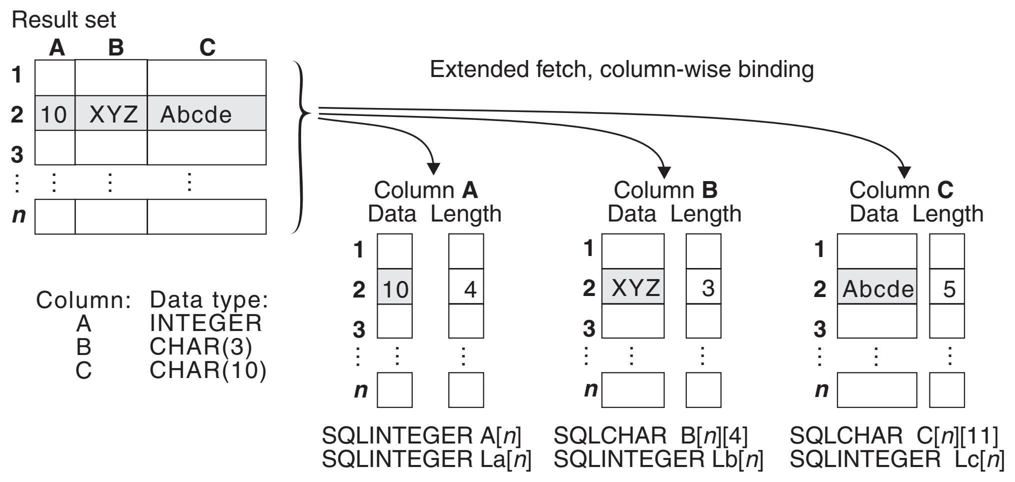


Figure 45. Column-wise binding

To perform column-wise array retrieval, include the following procedure in your application:

1. Call SQLSetStmtAttr() with the SQL_ATTR_ROWSET_SIZE attribute set to the number of rows that you want to retrieve with each fetch. When the value of the SQL_ATTR_ROWSET_SIZE attribute is greater than 1 on a statement handle, DB2 ODBC treats deferred output data pointers and length pointers of that handle as pointers to arrays.
2. Call SQLBindCol() for each column in the result set. In this call, include the following argument values:
   - Point the *rgbValue* argument to an array that is to receive data from the column that you specify with the *icol* argument.
   - For character and binary input data, specify the maximum size of the elements in the array with the input argument *cbValueMax*. (For other input data types, this argument is ignored.)

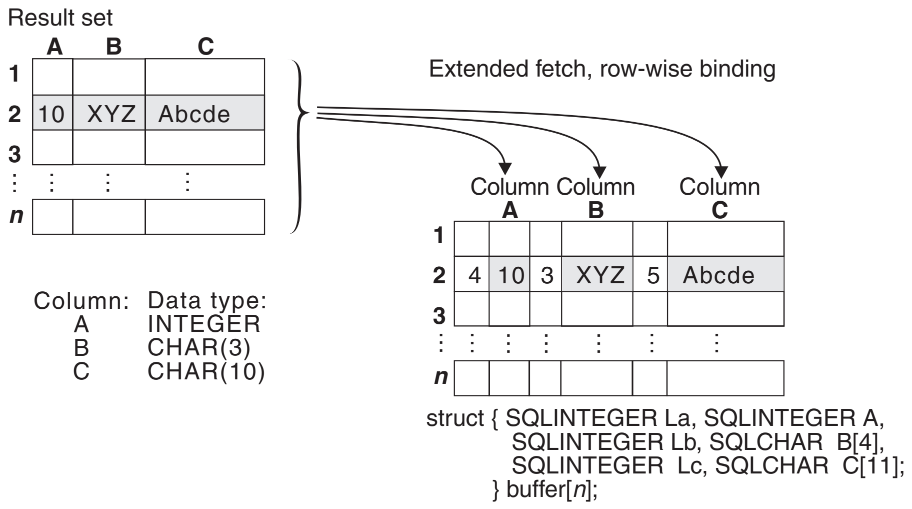
- Optionally, you can retrieve the number of bytes that each complete value requires in the array that is to receive the column data. To retrieve length data, point the *pcbValue* argument to an array that is to hold the number of bytes that DB2 ODBC will return for each retrieved value. Otherwise, you must set this value to NULL.

3. Call SQLExtendedFetch() to retrieve the result data into the array. If the number of rows in the result set is greater than the SQL_ATTR_ROWSET_SIZE attribute value, you must call SQLExtendedFetch() multiple times to retrieve all the rows.

DB2 ODBC uses the value of the maximum buffer size argument to determine where to store each successive result value in the array. You specify this value in the *cbValueMax* argument in SQLBindCol(). DB2 ODBC optionally stores the number of bytes that each element contains in a deferred length array. You specify this deferred array in the *pcbValue* argument in SQLBindCol().

Returning array data for row-wise bound data

Row-wise binding associates an entire row of the result set with a structure. You retrieve a row set that is bound in this manner into an array of structures. Each structure holds the data and associated length fields from an entire row. You use row-wise binding to retrieve data only, not to send it. Figure 46 gives a pictorial view of row-wise binding.



Result set
A
B
C
1
2
10
XYZ
Abcde
3
n
Extended fetch, row-wise binding
Column A
Column B
Column C
1
2
4
10
3
XYZ
5
Abcde
3
n
Column: Data type:
A INTEGER
B CHAR(3)
C CHAR(10)
struct { SQLINTEGER La, SQLINTEGER A,
SQLINTEGER Lb, SQLCHAR B[4],
SQLINTEGER Lc, SQLCHAR C[11];
} buffer[n];

Figure 46. Row-wise binding

To perform row-wise array retrieval, include the following procedure in your application:

1. Call SQLSetStmtAttr() with the SQL_ATTR_ROWSET_SIZE attribute to indicate how many rows to retrieve at a time.
2. Call SQLSetStmtAttr() again with the SQL_ATTR_BIND_TYPE attribute value set to the size of the structure to which the result columns are bound. When DB2 ODBC returns data, it uses the value of the SQL_ATTR_BIND_TYPE attribute to determine where to store successive rows in the array of structures.
3. Call SQLBindCol() to bind the array of structures to the result set. In this call, include the following argument values:

   - Point the *rgbValue* argument to the address of the element of the first structure in an array that is to receive data from the column that you specify with the *icol* argument.
   - For character and binary input data, specify the length, in bytes, of each element in the array that receives data in the input argument *cbValueMax*. (For other input data types, this argument is ignored.)
   - Optionally, point the *pcbValue* argument to the address of the element of the first structure in an array that is to receive the number of bytes that the column value for this bind occupies. Otherwise, set this value to NULL.
4. Call SQLExtendedFetch() to retrieve the data. If the number of rows in the result set is greater than the SQL_ATTR_ROWSET_SIZE attribute value, you must call SQLExtendedFetch() multiple times to retrieve all the rows.

Figure 47 shows the required functions to return column-wise and row-wise bound data. In this figure, *n* is the value of the SQL_ATTR_ROWSET_SIZE attribute and *m* is the number of columns in the result set. The left side of the figure shows how *n* rows are selected and retrieved one row at a time into *m* application variables where The right side of the figure shows how the same *n* rows are selected and retrieved directly into an array.

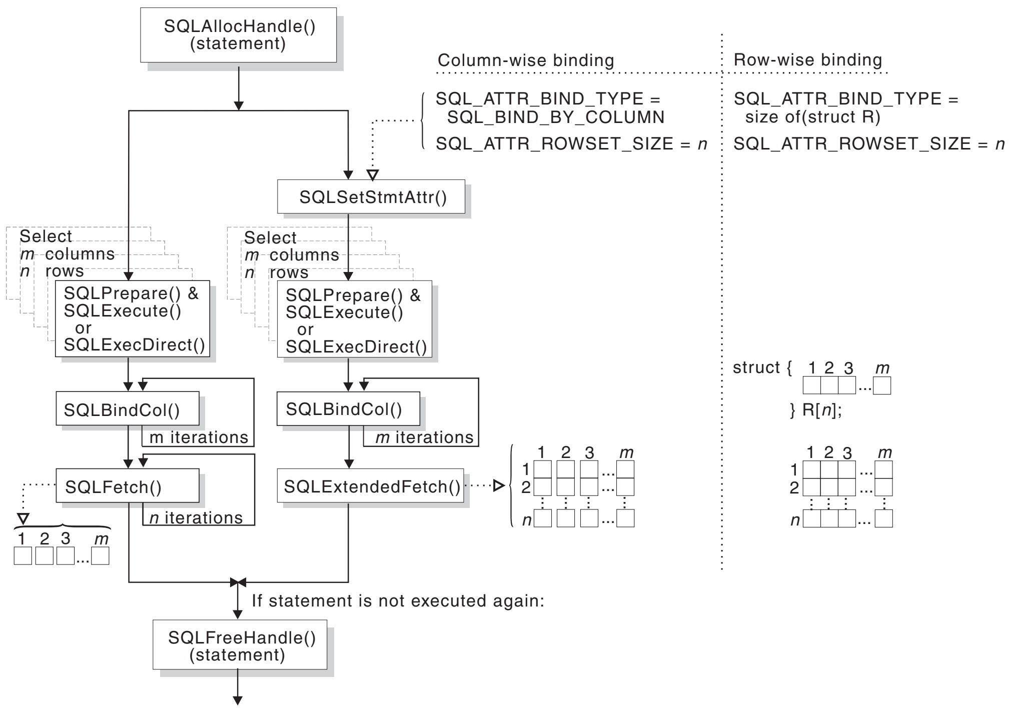


Figure 47. Array retrieval

Consider the following points when you perform array retrieval:

- If you specify the value *n* for SQL_ATTR_ROWSET_SIZE, you must retrieve the result set into an array of at least *n* elements. Otherwise memory overlay might occur.
- To bind *m* columns to application variables or an array, you must always make *m* calls to SQLBindCol().
- If the result set contains more rows than SQL_ATTR_ROWSET_SIZE specifies, you need to make multiple calls to SQLExtendedFetch() to retrieve all the rows in the result set. When you make multiple calls to SQLExtendedFetch(), you must

perform an operation between these calls to save the previously fetched data. These operations are listed in “Retrieving a result set into an array” on page 418.

Column-wise and row-wise binding example

Figure 48 shows an application that binds rows and columns of a result set to a structure.

```
/* ... */
#define NUM_CUSTOMERS 25
    SQLCHAR         stmt[] =
    {  "WITH "  /* Common Table expression (or Define Inline View) */
         "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 cust_num = CNUM(cast (? 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 "
     };
```

Figure 48. An application that retrieves data into an array by column and by row (Part 1 of 3)

```
/* 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
    };

#define  NUM_CUSTOMERS sizeof(Cust)/sizeof(SQLINTEGER)

/* Row-wise (Includes buffer for both column data and length) */
    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[ROWSET_SIZE];

    SQLUINTEGER     pirow = 0;
    SQLUINTEGER     pcrow;
    SQLINTEGER      i;
    SQLINTEGER      j;
/* ... */
    /* Get details and total for each order row-wise */
    rc = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

    rc = SQLParamOptions(hstmt, NUM_CUSTOMERS, &pirow);

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

    rc = SQLExecDirect(hstmt, stmt, SQL_NTS);

    /* SQL_ROWSET_SIZE sets the max number  */
    /* of result rows to fetch each time    */
    rc = SQLSetStmtAttr(hstmt, SQL_ATTR_ROWSET_SIZE,
                        (void *)ROWSET_SIZE, 0);

    /* Set size of one row, used for row-wise binding only */
    rc = SQLSetStmtAttr(hstmt, SQL_ATTR_BIND_TYPE,
                        (void *)sizeof(Ord) / ROWSET_SIZE, 0);
```

Figure 48. An application that retrieves data into an array by column and by row (Part 2 of 3)

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

    /* Bind remaining columns ... */
/* ... */

   /* NOTE: This sample assumes that an order never has more
            rows than ROWSET_SIZE. A check should be added below to call
            SQLExtendedFetch multiple times for each result set.
    */
    do  /* for each result set .... */
    { rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, NULL);

      if (pcrow > 0) /* if 1 or more rows in the result set */
      {
        i = j = 0;

        printf("**************************************\n");
        printf("Orders for Customer: %ld\n", Ord[0].Cust_Num);
        printf("**************************************\n");

        while (i < pcrow)
        {   printf("\nOrder #: %ld\n", Ord[i].Ord_Num);
            printf("     Product  Quantity         Price\n");
            printf("     -------- ---------------- ------------\n");
            j = i;
            while (Ord[j].Ord_Num == Ord[i].Ord_Num)
            {   printf("    %8ld %16.7lf %12.2lf\n",
                       Ord[i].Prod_Num, Ord[i].Quant, Ord[i].Amount);
                i++;
            }
            printf("                              ============\n");
            printf("                             %12.2lf\n", Ord[j].Total);
        } /* end while */
      } /* end if */
    }
    while ( SQLMoreResults(hstmt) == SQL_SUCCESS);
/* ... */
```

Figure 48. An application that retrieves data into an array by column and by row (Part 3 of 3)

Using large objects

The term large object and the generic acronym LOB refer to any type of large object. DB2 supports the following three LOB data types:
- Binary large object (BLOB)
- Character large object (CLOB)
- Double-byte character large object (DBCLOB)

These LOB data types are represented symbolically as SQL_BLOB, SQL_CLOB, SQL_DBCLOB respectively. All DB2 ODBC functions that accept or return SQL data type arguments (for example, the SQLBindParameter() and SQLDescribeCol() functions) can accept or return LOB symbolic constants. See Table 4 on page 25 for a complete list of symbolic and default C symbolic names for SQL data types.

An application can retrieve and manipulate LOB values in the application address space. However, your application might not require you to transfer the entire LOB from the database server into application memory. In many cases, you can select a

LOB value and operate on pieces of it. The ODBC model can transfer LOB data using the piecewise sequential method with SQLGetData() and SQLPutData(). This method might prove inefficient. You can more efficiently retrieve and manipulate an individual LOB value by using a LOB locator.

Using LOB locators

LOB locators enable you to identify and manipulate LOB values at the database server. They also enable you to retrieve only pieces of a LOB value into application memory.

Locators are a run-time concept: they are not a persistent type, nor are they stored in the database. Conceptually, LOB locators are simple token values (much like a pointer) that you use to refer to much larger LOB values in the database. LOB locator values do not persist beyond the transaction in which they are created (unless you specify otherwise).

A locator references a LOB value, not the physical location (or address) at which a LOB value resides. The LOB value that a locator references does not change if the original LOB value in the table is altered. When you perform operations on a locator, these operations similarly do not alter the original LOB value that the table contains. To materialize operations that you perform on LOB locators, you must store the result of these operations in a location on the database server, or in a variable within your application.

In DB2 ODBC functions, you specify LOB locators with one of the following symbolic C data types:

- SQL_C_BLOB_LOCATOR for BLOB data
- SQL_C_CLOB_LOCATOR for CLOB data
- SQL_C_DBCLOB_LOCATOR for DBCLOB data

Choose a C type that corresponds to the LOB data to which you refer with the locator. Through these C data types, you can transfer a small token value to and from the database server instead of an entire LOB value.

Call SQLBindCol() and SQLFetch(), or SQLGetData() to retrieve a LOB locator that is associated with a LOB value into an application variable. You can then apply the following DB2 ODBC functions to that locator:

- SQLGetLength(), which returns the length of the string that a LOB locator represents.
- SQLGetPosition(), which returns the position of a search string within a source string that a LOB locator represents. LOB locators can represent both search strings and source strings.

The following actions implicitly allocate LOB locators:

- 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; otherwise an error occurs.

You can also use LOB locators to move LOB data at the server without pulling data into application memory and then sending it back to the server.

Example: The following INSERT SQL statement concatenates two LOB values with LOB locators (which are represented by the parameter markers) and inserts the result into a table:

```
INSERT INTO TABLE4A
VALUES(1,CAST(? AS CLOB(2K)) CONCAT CAST(? AS CLOB(3K)))
```

You can explicitly free a locator before the end of a transaction with the FREE LOCATOR statement. You can explicitly retain a locator beyond a unit of work with the HOLD LOCATOR statement. You execute these statements with the following syntax:

```
                      ┌─,─────────────┐
                      ▼               │
►►──FREE LOCATOR────host_variable─┴──────────────────────────────►◄
```

```
                      ┌─,─────────────┐
                      ▼               │
►►──HOLD LOCATOR────host_variable─┴──────────────────────────────►◄
```

Although you cannot prepare the FREE LOCATOR SQL statement or the HOLD LOCATOR SQL statement dynamically, DB2 ODBC accepts these statements in SQLPrepare() and SQLExecDirect(). Use parameter markers in these statements so that you can convert application variables that contain LOB locator values to host variables that these SQL statements can access. Before you call SQLPrepare() or SQLExecDirect(), call SQLBindParameter() with the data type arguments set to the appropriate SQL and C symbolic data types. (See Table 4 on page 25 for a list of these data types.) This call to SQLBindParameter() passes an application variable that contains the locator value into the parameter markers as a host variable.

LOB locators and functions that are associated with locators (such as the SQLGetSubString(), SQLGetPosition(), and SQLGetLength() functions) are not available when you connect to a DB2 server that does not support large objects. To determine if a connection supports LOBs, call SQLGetFunctions() with the function type set to SQL_API_SQLGETSUBSTRING. If the *pfExists* output argument returns SQL_TRUE, the current connection supports LOBs. If the *pfExists* output argument returns SQL_FALSE, the current connection does not support LOBs.

LOB and LOB locator example

Figure 49 on page 426 shows an example application that extracts the 'Interests' section from the RESUME CLOB column of the EMP_RESUME table. This application transfers only a substring into memory.

```
/* ... */
    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_INPUT, SQL_C_CHAR, SQL_CHAR, 7,
                           0, Empno.s, sizeof(Empno.s), &Empno.ind);

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

    rc = SQLExecDirect(hstmt, stmt2, SQL_NTS);
    rc = SQLBindCol(hstmt, 1, SQL_C_CLOB_LOCATOR, &ClobLoc1, 0,
                      &pcbValue);
    rc = SQLFetch(hstmt);
/******************************************************************
    Search CLOB locator to find "Interests"
    Get substring of resume (from position of interests to end)
*******************************************************************/

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

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

    /* Get starting postion */
    rc = SQLGetPosition(lhstmt, SQL_C_CLOB_LOCATOR, ClobLoc1, 0,
                    "Interests", 9, 1, &Pos1, &Ind);


    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);

    /* Print Interest section of Employee's resume */
    printf("\nEmployee #: %s\n %s\n", Empno.s, buffer);
/* ... */
```

Figure 49. An application that uses LOB locators

Using distinct types

In addition to the built-in SQL data types, you can define your own SQL data type, which are called *distinct types*. When you create a distinct type, you base it on an existing SQL built-in type. This SQL built-in type is called the *source type*. Internally, a distinct type and the source type are equivalent, but for most programming operations a distinct type is incompatible with the source type. You create distinct types with the CREATE DISTINCT TYPE SQL statement. See *DB2 SQL Reference* for more information about this statement.

Distinct types help provide the strong typing control that an object-oriented program requires. When you use distinct types, you ensure that only functions and operators that are explicitly defined on a distinct type can be applied to instances of that type.

When you use distinct types, applications continue to work with C data types for application variables. You need to consider only the distinct types when you construct SQL statements.

The following guidelines apply to distinct types:

- All SQL-to-C data type conversion rules that apply to the source type also apply to the distinct type.
- The distinct type has the same default C type as the source type.
- SQLDescribeCol() returns the source type for distinct type columns. Call SQLColAttribute() with the input descriptor type set to SQL_DESC_DISTINCT_TYPE to obtain distinct type names.
- When you use an SQL predicate that compares a distinct type to a parameter marker, you must either cast the parameter marker to the distinct type or cast the distinct type to a source type. This casting is required because distinct types are not compatible with other data types in comparison operations. Applications use only C data types that represent SQL built-in types. This difference between C types and SQL types requires you to cast from the C built-in type to the SQL distinct type within the SQL statement. Alternatively you can cast the distinct type to a source type, which C types support. If you do not make one of these conversions, an error occurs when you prepare the statement. For more information about casting distinct types, see "Distinct types" on page 471.

Figure 50 shows an application that creates distinct types, user-defined functions, and tables with distinct type columns. For an example that inserts rows into a table with distinct type columns, see Figure 44 on page 417.

```
/* ... */
    /* Initialize SQL statement strings  */
    SQLCHAR         stmt[][MAX_STMT_LEN] = {
        "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)",
```

Figure 50. An application that creates distinct types (Part 1 of 2)

```
     "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 )"
    };
/* ... */
    num_stmts = sizeof(stmt) / MAX_STMT_LEN;

    printf(">Executing %ld Statements\n", num_stmts);

    /* Execute Direct statements */
    for (i = 0; i < num_stmts; i++) {
        rc = SQLExecDirect(hstmt, stmt[i], SQL_NTS);
    }
/* ... */
```

Figure 50. An application that creates distinct types (Part 2 of 2)

Using stored procedures

You can design an application to run in two parts: one part on the client and one part on the server. Stored procedures are server applications that run at the database, within the same transaction as a client application. You can write stored procedures with either embedded SQL or DB2 ODBC functions. (See “Writing a DB2 ODBC stored procedure” on page 430 for more information about stored procedures.)

Both the main application that calls a stored procedure and a stored procedure itself can be either a DB2 ODBC application or a standard DB2 precompiled application. You can use any combination of embedded SQL and DB2 ODBC applications. Figure 51 illustrates this concept.

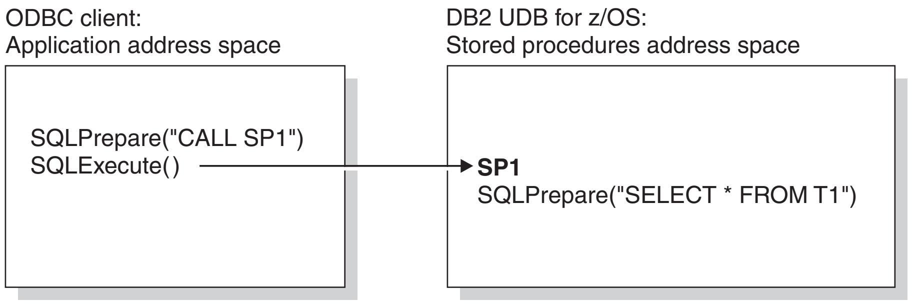

ODBC client:
Application address space
DB2 UDB for z/OS:
Stored procedures address space
SQLPrepare("CALL SP1")
SQLExecute()
SP1
SQLPrepare("SELECT * FROM T1")

Figure 51. Running stored procedures

Advantages of using stored procedures

In general, stored procedures provide the following advantages:

- You avoid network transfer of large amounts of data obtained as part of intermediate results in a long sequence of queries.
- You deploy client database applications into client/server pieces.

Stored procedures written in embedded static SQL have the following additional 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 database objects in order to access them. Static SQL can help enforce this encapsulation.
- Security: Users’ access privileges are encapsulated within the packages associated with the stored procedures, so you do not need to grant explicit access to each database object. For example, you can grant a user run access for a stored procedure that selects data from tables for which the user does not have select privilege.

Catalog table for stored procedures

Registered stored procedures are defined in the SYSIBM.SYSROUTINES catalog table. Call SQLProcedureColumns() to determine the input and output parameters that are associated with a procedure call. For more information about this function, see “SQLProcedureColumns() - Get procedure input/output parameter information” on page 320.

Calling stored procedures from a DB2 ODBC application

To invoke stored procedures from a DB2 ODBC application, pass a CALL statement with the following syntax to SQLExecDirect(), or to SQLPrepare() followed by SQLExecute().

```
                                  ┌─,───┐
                                  ▼     │
►►──CALL──procedure-name──(─────┬───┬─┴──)──────────────────────────────►◄
                                  └─?─┘
```

procedure-name
: The name of the stored procedure to execute. Call SQLProcedures() to obtain a list of stored procedures that are available at the database.

Although the CALL statement cannot be prepared dynamically, DB2 ODBC accepts the CALL statement as if it can be dynamically prepared. You can also call stored procedures with the ODBC vendor escape sequence. For more information about calling a stored procedure with an ODBC vendor escape sequence, see “Stored procedure CALL” on page 469.

The question mark (?) in the CALL statement syntax diagram denotes parameter markers that correspond to the arguments for a stored procedure. You must pass all arguments to a stored procedure with parameter markers. Literals, the NULL keyword, and special registers are not allowed. However, you can use literals if you include a vendor escape clause in your CALL statement. See “Using vendor escape clauses” on page 465 for more information about including literals in a CALL statement.

You bind the parameter markers in a CALL statement to application variables with SQLBindParameter(). Although you can use stored procedure arguments that are both input and output arguments, you should avoid sending unnecessary data between the client and the server. Specify either SQL_PARAM_INPUT for input arguments or SQL_PARAM_OUTPUT for output arguments when you call SQLBindParameter(). Specify SQL_PARAM_INPUT_OUTPUT only if the stored procedure uses arguments that are both input and output arguments. Literals are considered type SQL_PARAM_INPUT only.

For more information about the use of the CALL statement and stored procedures, see *DB2 SQL Reference* and *DB2 Application Programming and SQL Guide*.

Writing a DB2 ODBC stored procedure

Although stored procedures that are written in embedded SQL provide more advantages than stored procedures that are written in ODBC, you might want components of DB2 ODBC applications to run on servers. You can write stored procedures in DB2 ODBC to minimize the required changes to the code and logic of those components.

You write ODBC stored procedures as ordinary ODBC applications, with the following exceptions:

- You must turn off AUTOCOMMIT. Set the SQL_ATTR_AUTOCOMMIT attribute to SQL_AUTOCOMMIT_OFF with SQLSetConnectAttr(). You can also specify AUTOCOMMIT=0 in the DB2 ODBC initialization file to disable AUTOCOMMIT.

- You must make a null database connection with SQLConnect(). A stored procedure runs under the same connection and transaction as the client application. A null SQLConnect() call associates a connection handle in the stored procedure with the underlying connection of the client application. To make a null SQLConnect() call, set the *szDSN*, *szUID*, and *szAuthStr* argument pointers to NULL, and set their respective length arguments to 0.
- If your stored procedure contains any LOB data types or distinct types in its parameter list, specify MVSATTACHTYPE=RRSAF in the DB2 ODBC initialization file. DB2 UDB for z/OS requires that stored procedures containing any LOBs or distinct types must run in a WLM-established stored procedure address space.

When you define a DB2 ODBC stored procedure to DB2, specify the COMMIT ON RETURN NO clause in the CREATE PROCEDURE SQL statement. For stored procedures that are written in DB2 ODBC, the COMMIT ON RETURN clause has no effect on DB2 ODBC rules. However, COMMIT ON RETURN NO overrides the manual-commit mode that is set in the client application. For more information about setting up the stored procedures environment, see Part 6 of *DB2 Application Programming and SQL Guide*.

Returning result sets from stored procedures

In DB2 ODBC applications, you use open cursors to retrieve result sets from stored procedure calls. Stored procedures that return result sets to DB2 ODBC open one or more cursors that are each associated with a query, and keep these cursors open when the stored procedure exits. When a stored procedure leaves more than one cursor open after it exits, it returns multiple result sets.

When you define a stored procedure that returns result sets, you must specify the maximum number of result sets that the procedure is to return. You specify this value in the DYNAMIC RESULT SETS clause in the CREATE PROCEDURE SQL statement. This value appears in the RESULT_SETS column of the SYSIBM.SYSROUTINES table for all stored procedures. A zero in this column indicates that open cursors return no result sets. Zero is the default value.

Programming stored procedures to return result sets

In general, you write a stored procedure that returns result sets to a DB2 ODBC application to perform the following actions:

- For each result set the stored procedure returns, declare a cursor with the WITH RETURN option, open the cursor on the result set (that is, execute a query), and leave the cursor open after you exit the procedure.
- Return a result set for every cursor that is left open after exit, in the order in which the procedure opened the corresponding cursors.
- Pass only unread rows back to the DB2 ODBC client application.

  For example, if the result set of a cursor has 500 rows, but the stored procedure reads 150 of those rows before it terminates, the stored procedure returns only rows 151 through 500. You can use this behavior to filter out initial rows in the result set before you return them to the client application.

More specifically, to write a DB2 ODBC stored procedure that returns result sets, you must include the following procedure in your application:

1. Issue SQLExecute() or SQLExecDirect() to perform a query that opens a cursor. In stored procedures, DB2 ODBC declares cursors with the WITH RETURN option.
2. Optionally, issue SQLFetch() to read rows that you want to filter from the result set.

3. Issue SQLDisconnect(), SQLFreeHandle() with *HandleType* set to SQL_HANDLE_DBC, and SQLFreeHandle() with *HandleType* set to SQL_HANDLE_ENV to exit the stored procedure. This exit leaves the statement handle, and the corresponding cursor, in a valid state.

Do not issue SQLFreeHandle() with *HandleType* set to SQL_HANDLE_STMT or SQLCloseCursor(). When you do not free the statement handle or explicitly close the cursor on that handle, the cursor remains open to return result sets. If you close a cursor before the stored procedure exit, it is a local cursor. If you keep a cursor open after you exit the stored procedure, it returns a query result set (also called a multiple result set) to the client application. Appendix F, "Example DB2 ODBC code," on page 531 provides an example; see case 2 of step 4.

Restrictions on stored procedures returning result sets

In general, when you call a stored procedure that returns a result set, it is equivalent to executing a query statement. The following restrictions apply to this equivalency:

- SQLDescribeCol() or SQLColAttribute() do not return column names for static query statements. In this case of static statements, these functions return the ordinal position of columns instead.
- All result sets are read-only.
- You cannot use schema functions (such as the SQLTables() function) to return a result set. If you use schema functions within a stored procedure, you must close all cursors that are associated with the statement handles of those functions. If you do not close these cursors, your stored procedure might return extraneous result sets.
- When you prepare a stored procedure, you cannot access the column information for the result set until after you issue the CALL statement. Normally, you can access result set column information immediately after you prepare a query.

Programming DB2 ODBC client applications to receive result sets

After you execute a stored procedure from a client application, you receive the result sets from that stored procedure in the way that you receive result sets from a query. To write a DB2 ODBC client application that receives result sets from a stored procedure, perform the following actions in your application:

1. Ensure that no open cursors are associated with the statement handle on which you plan to issue the CALL SQL statement.
2. Call SQLPrepare() and SQLExecute(), or call SQLExecDirect() to issue the CALL SQL statement for the stored procedure that you want to invoke. This execution of the CALL SQL statement effectively causes the cursors that are associated with the result sets to open.
3. Examine output parameters that the stored procedure returns. For example, the procedure might be designed with an output parameter that indicates exactly how many result sets are generated. You could then use this information to receive those result sets more efficiently.
4. If you do not know the nature of the result set, or the number of columns that the result set is to contain, call SQLNumResultCols(), SQLDescribeCol(), or SQLColAttribute().
5. Use any permitted combination of SQLBindCol(), SQLFetch(), and SQLGetData() to obtain the data set from the current cursor. You must process result sets serially. You receive each result set one at a time in the order that the stored procedure opens the corresponding cursors.

6. When you finish processing the current result set, call SQLMoreResults() to check for more result sets to receive. If an additional result set exists, SQLMoreResults() returns SQL_SUCCESS, closes the current cursor, and advances processing to the next open cursor. Otherwise, SQLMoreResults() returns SQL_NO_DATA_FOUND. Repeat steps 3 on page 432 through 6 until you receive all result sets that the stored procedure returned.

Stored procedure example with query result set

A detailed stored procedure example is provided in Appendix F, “Example DB2 ODBC code,” on page 531.

Writing multithreaded and multiple-context applications

This section explains DB2 ODBC multithread and multiple context support, and it provides guidelines about how you use contexts and threads together in an application.

DB2 ODBC support for multiple Language Environment threads

A Language Environment thread represents an independent instance of a routine within an application. When you execute a DB2 ODBC application, it begins with an initial Language Environment thread, or parent thread. To make your application multithreaded, call the POSIX Pthread function pthread_create() within your application. This function creates additional Language Environment threads, or child threads, which work concurrently with the parent thread.

You must run multithreaded DB2 ODBC applications in one of the following environments:

- The z/OS UNIX environment.
- For applications that are HFS-resident, TSO or batch environments that use the IBM-supplied BPXBATCH program. See *z/OS UNIX System Services Command Reference* for more information about BPXBATCH.
- For applications that are not HFS-resident, TSO or batch environments that use the Language Environment run-time option POSIX(ON). See *z/OS Language Environment Programming Guide* for more information about running POSIX-enabled programs.

  Example: To run the multithreaded, non-HFS, DB2 ODBC application APP1 in the data set USER.RUNLIB.LOAD, you could use one of the following approaches:

  – Use TSO to enter the command:

  ```
  CALL 'USER.RUNLIB.LOAD(APP1)' 'POSIX(ON)/'
  ```

  – Use batch JCL to submit the job:

  ```
  //STEP1 EXEC PGM=APP1,PARM='POSIX(ON)/'
  //STEPLIB DD DSN=USER.RUNLIB.LOAD,DISP=SHR
  //        DD ...other libraries needed at run time...
  ```

The collection of all the Language Environment threads in an application make an independent set of routines called a Language Environment *enclave*. All Language Environment threads within an enclave share the same reentrant copy of the DB2 ODBC driver code. DB2 ODBC must also protect shared storage when multiple Language Environment threads run concurrently in the same enclave. Reentrant code that correctly handles shared storage is referred to as *threadsafe*. Multithreaded ODBC applications require a threadsafe driver.

The DB2 ODBC driver is threadsafe. DB2 ODBC supports the concurrent execution of Language Environment threads. Your DB2 ODBC applications will support multiple Language Environment threads, only if the following conditions are true:

- DB2 ODBC can access the z/OS UNIX environment. DB2 ODBC uses Pthread *mutex* functions, which the z/OS UNIX environment provides, to serialize critical sections of DB2 ODBC code. With these Pthread mutex functions, all DB2 ODBC functions are threadsafe. For more information about the z/OS UNIX environment in relation to DB2 ODBC, see "Setting up the z/OS UNIX environment" on page 44, and "Preparing and executing a DB2 ODBC application" on page 44.
- THREADSAFE=0 is not specified in the initialization file. You can use the THREADSAFE keyword to specify whether the DB2 ODBC driver uses Pthread mutex functions to make your applications threadsafe. See "Initialization keywords" on page 51 for a description of the THREADSAFE keyword.

Multithreaded applications use threads to perform work in parallel. Figure 52 depicts an application that performs parallel operations on two different connections and manages a shared application buffer.

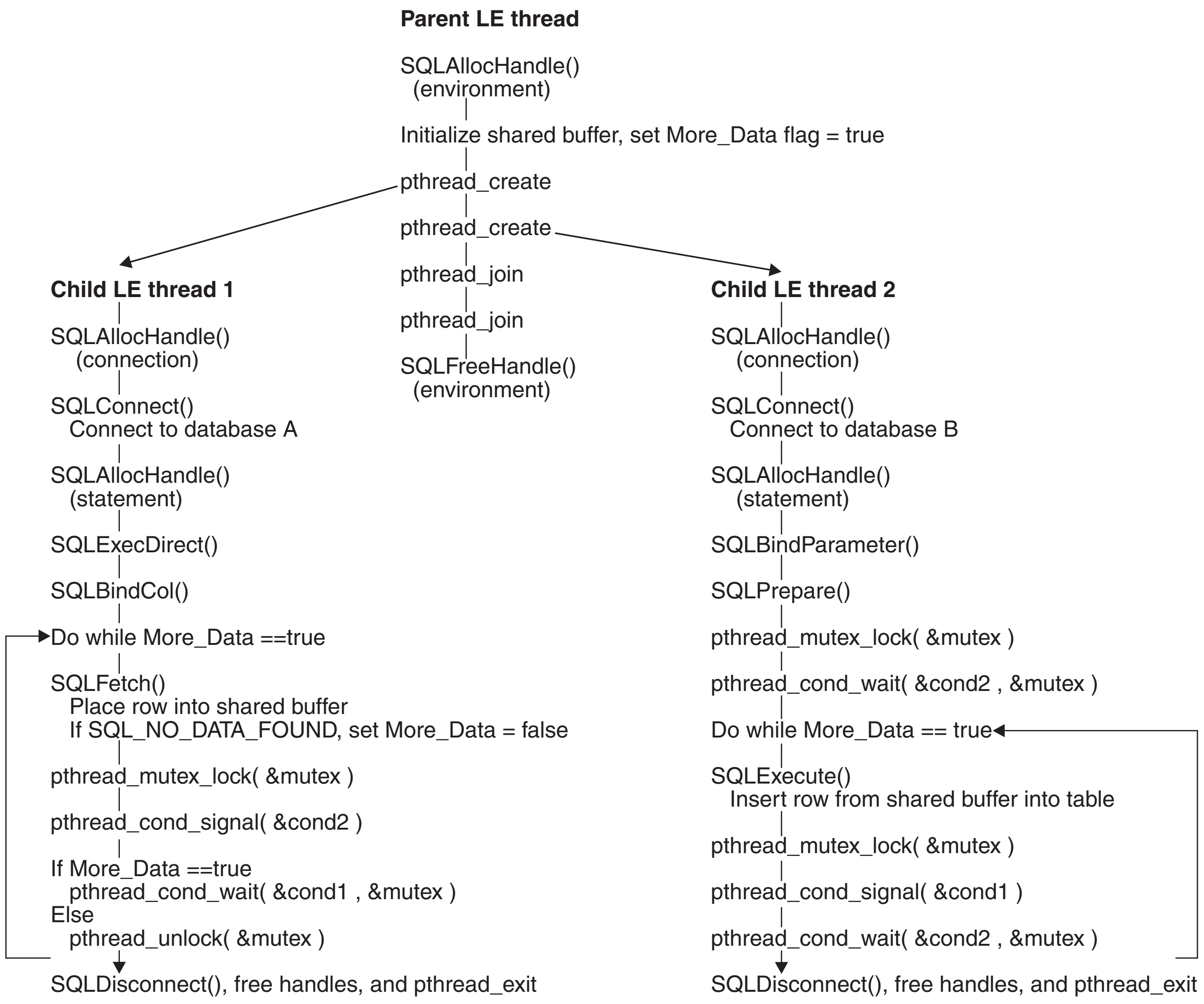

Parent LE thread
SQLAllocHandle() (environment)
Initialize shared buffer, set More_Data flag = true
pthread_create
pthread_create
pthread_join
pthread_join
SQLFreeHandle() (environment)
Child LE thread 1
SQLAllocHandle() (connection)
SQLConnect() Connect to database A
SQLAllocHandle() (statement)
SQLExecDirect()
SQLBindCol()
Do while More_Data ==true
SQLFetch() Place row into shared buffer If SQL_NO_DATA_FOUND, set More_Data = false
pthread_mutex_lock(&mutex)
pthread_cond_signal(&cond2)
If More_Data ==true pthread_cond_wait(&cond1 , &mutex) Else pthread_unlock(&mutex)
SQLDisconnect(), free handles, and pthread_exit
Child LE thread 2
SQLAllocHandle() (connection)
SQLConnect() Connect to database B
SQLAllocHandle() (statement)
SQLBindParameter()
SQLPrepare()
pthread_mutex_lock(&mutex)
pthread_cond_wait(&cond2 , &mutex)
Do while More_Data == true
SQLExecute() Insert row from shared buffer into table
pthread_mutex_lock(&mutex)
pthread_cond_signal(&cond1)
pthread_cond_wait(&cond2 , &mutex)
SQLDisconnect(), free handles, and pthread_exit

Figure 52. Multithreaded application

The application that Figure 52 on page 434 portrays an application that performs the following steps to make a parallel database-to-database copy:

1. Creates two child Language Environment threads from an initial parent thread. The parent thread remains active for the duration of the child threads. DB2 ODBC requires that the thread that establishes the environment handle must persist for the duration of the application. The persistence of this thread keeps DB2 language interface routines resident in the Language Environment enclave.
2. Connects to database A with child Language Environment thread 1 and uses SQLFetch() to read data from this connection into a shared application buffer.
3. Connects to database B with child Language Environment thread 2. Child Language Environment thread 2 concurrently reads data from the shared application buffer and inserts this data into database B.
4. Calls Pthread functions to synchronize the use of the shared application buffer within each of the child threads. See *z/OS C/C++ Run-Time Library Reference* for a description of the Pthread functions.

When to use multiple Language Environment threads

A detailed discussion about when to use multithreading in your application is beyond the scope of this book. However, some general application types are well-suited to multithreading. For example, applications that handle asynchronous work requests make good candidates for multithreading.

An application that handles asynchronous work requests can take the form of a parent-child threading model in which the parent Language Environment thread creates child Language Environment threads to handle incoming work. The parent thread can then dispatch these work requests, as they arrive, to child threads that are not currently busy handling other work.

For more information about when to use threads in your DB2 ODBC applications, see *z/OS C/C++ Programming Guide*.

DB2 ODBC support of multiple contexts

A *context* is the DB2 ODBC equivalent of a DB2 thread. Contexts are the structures that describe the logical connections that an application makes to data sources and the internal DB2 ODBC connection information that allows applications to direct operations to a data source. You establish a context when you allocate a connection handle when multiple contexts are enabled.

DB2 ODBC always creates a context for the first connection handle that you create on a Language Environment thread. If you do not enable DB2 ODBC support for multiple contexts, only these SQLAllocHandle() calls establish a context. If you enable support for multiple contexts, DB2 ODBC establishes a separate context (and DB2 thread) each time that you issue SQLAllocHandle() to allocate a connection handle.

To enable or explicitly disable DB2 ODBC support for multiple contexts, use the MULTICONTEXT keyword in the DB2 ODBC initialization file.

Before you enable multiple contexts, each Language Environment thread that you create can use only a single context. With only one context for each Language Environment thread, your application runs with only simulated support for the ODBC connection model. Multiple contexts are disabled by default. To explicitly disable multiple contexts, specify MULTICONTEXT=0 in the initialization file. For more information about the ODBC connection model, see "DB2 ODBC restrictions on the ODBC connection model" on page 11.

When you specify MULTICONTEXT=1 in the initialization file, a distinct context is established for each connection handle, which you establish with SQLAllocHandle(). With a context for each connection, DB2 ODBC is consistent with, and provides full support for, the ODBC connection model.

To use multiple contexts, you must specify MVSATTACHTYPE=RRSAF in the initialization file.

Specifying MULTICONTEXT=1 implies CONNECTTYPE=1. Implicitly concurrent connection types are consistent with the ODBC connection model. SQLEndTran() handles all connections independently for both commit and rollback. See "Initialization keywords" on page 51 for more information about CONNECTTYPE and MULTICONTEXT.

In a multiple-context environment, you establish contexts with SQLAllocHandle() and delete contexts with SQLFreeHandle() (with the *HandleType* argument on both functions set to SQL_HANDLE_DBC). All SQLConnect() and SQLDisconnect() operations that use the same connection handle belong to the same context. Although you can make only one active connection to a data source within a single context, you can call SQLDisconnect() and then call SQLConnect() to change the target data source. When you change data sources in a multiple-context environment, this change is also subject to the rules of CONNECTTYPE=1.

When you specify MULTICONTEXT=1, DB2 ODBC automatically uses z/OS Unauthorized Context Services to create and manage contexts for the application. However, DB2 ODBC does not perform context management for the application if any of the following conditions are true:

- Your DB2 ODBC application creates a DB2 thread before it invokes DB2 ODBC. This condition always applies for any stored procedure that uses DB2 ODBC.
- Your DB2 ODBC application creates and switches to a private context before it invokes DB2 ODBC. For example, an application that explicitly uses z/OS Unauthorized Context Services and that issues ctxswch() to switch to a private context prior to invoking DB2 ODBC cannot take advantage of MULTICONTEXT=1.
- Your DB2 ODBC application starts a unit of recovery with any RRS resource manager before it invokes DB2 ODBC.
- You specify MVSATTACHTYPE=CAF in the initialization file.
- The operating system level does not support Unauthorized Context Services.

To determine if MULTICONTEXT=1 is active for the DB2 ODBC application, call SQLGetInfo() with the *InfoType* argument set to SQL_MULTIPLE_ACTIVE_TXN. See "SQLGetInfo() - Get general information" on page 234 for a full description of SQLGetInfo().

Table 228 on page 437 shows the connection characteristics that different combinations of MULTICONTEXT and CONNECTTYPE produce.

Table 228. Connection characteristics

Settings		Results		
MULTICONTEXT	**CONNECTTYPE**	**Language Environment threads can have more than one ODBC connection with an outstanding unit of work**	**Language Environment threads can commit or rollback an ODBC connection independently**	**Number of DB2 threads that DB2 ODBC creates on behalf of application**
0	2	Y	N	1 per Language Environment thread
0	1	N	Y	1 per Language Environment thread
1[1]	1 or 2[2]	Y	Y	1 per ODBC connection handle

Notes:

1. MULTICONTEXT=1 requires MVSATTACHTYPE=RRSAF
2. MULTICONTEXT=1 implies CONNECTTYPE=1 characteristics. If you specify MULTICONTEXT=1 and CONNECTTYPE=2 in the initialization file, DB2 ODBC ignores CONNECTTYPE=2. When you specify MULTICONTEXT=1, any attempt to set CONNECTTYPE=2 with SQLSetEnvAttr(), SQLSetConnectAttr(), or SQLDriverConnect() is rejected with SQLSTATE **01**S02.
   - All connections in a DB2 ODBC application have the same CONNECTTYPE and MULTICONTEXT characteristics. The connection type of an application (which is specified with the CONNECTTYPE keyword) is established at the first SQLConnect() call. Multiple-context support (which is specified with the MULTICONTEXT keyword) is established when you allocate an environment handle.
   - For CONNECTTYPE=1 or MULTICONTEXT=1, the AUTOCOMMIT default value is ON. For CONNECTTYPE=2 or MULTICONTEXT=0, the AUTOCOMMIT default value is OFF.

Multiple contexts, one Language Environment thread

When you specify the initialization file setting MULTICONTEXT=1, a DB2 ODBC application can create multiple independent connections for each Language Environment thread. Figure 53 on page 438 is an example of an application that uses multiple contexts on one Language Environment thread.

```
/*  Get an environment handle (henv).            */

SQLAllocHandle(SQL_HANDLE_ENV, SQL_HANDLE_NULL, &henv );

/*
 *  Get two connection handles, hdbc1 and hdbc2, which
 *  represent two independent DB2 threads.
 */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc1 );
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc2 );

/*  Set autocommit off for both connections.     */
/*  This is done only to emphasize the           */
/*  independence of the connections for purposes */
/*  of this example, and is not intended as      */
/*  a general recommendation.                    */

SQLSetConnectAttr(hdbc1, SQL_ATTR_AUTOCOMMIT, (void *)SQL_AUTOCOMMIT_OFF, 0 );
SQLSetConnectAttr(hdbc2, SQL_ATTR_AUTOCOMMIT, (void*)SQL_AUTOCOMMIT_OFF, 0 );

/*  Perform SQL under DB2 thread 1 at STLEC1.    */

SQLConnect( hdbc1, (SQLCHAR *) "STLEC1", ... );
SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
SQLExecDirect ...
      .
      .
/*  Perform SQL under DB2 thread 2 at STLEC1.    */

SQLConnect( hdbc2, (SQLCHAR *) "STLEC1", ... );
SQLAllocHandle(SQL_HANDLE_STMT, hdbc2, &hstmt2);
SQLExecDirect ...
     .
     .
/*  Commit changes on connection 1.              */

SQLEndTran(SQL_HANDLE_DBC, hdbc1, SQL_COMMIT);

/*  Rollback changes on connection 2.            */

SQLEndTran(SQL_HANDLE_DBC, hdbc2, SQL_ROLLBACK);
     .
     .
```

Figure 53. An application that makes independent connections on a single Language Environment thread

Multiple contexts, multiple Language Environment threads

When you combine the initialization file setting MULTICONTEXT=1 with the default setting THREADSAFE=1, your application can create multiple independent connections under multiple Language Environment threads. With this capability, you can use a fixed number of Language Environment threads to implement complex DB2 ODBC server applications that handle multiple incoming work requests.

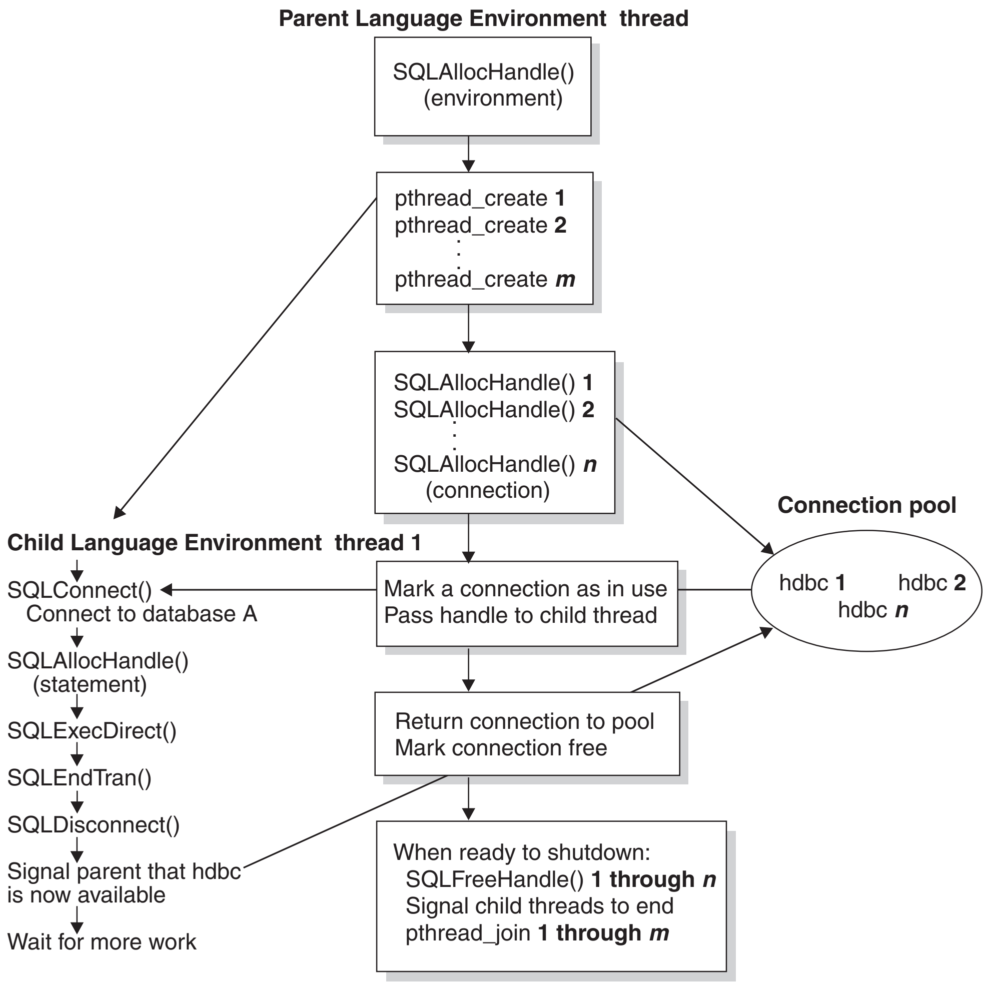
Applications that use both multiple contexts and multiple Language Environment threads require you to manage application resources. Use the Pthread functions or another internal mechanism to prevent different threads from using the same connection handles or statement handles. Figure 54 on page 439 shows how an application can fail without a mechanism to serialize use of handles.

```
LE_Thread_1                        LE_Thread_2
  .                                  .
  .                                  .
  .                                  .
  .                                  .
  .                                  .
rc = SQLExecDirect( hstmt1 , ... );  .
  .                                  .
  .                                  .
  .                                SQLFreeHandle( hstmt1 , SQL_DROP);
  .                                  .
  .                                  .
  .                                  .

SQLExecDirect() returns SQL_INVALID_HANDLE
because LE_Thread_2 frees hstmt1 before LE_Thread_1
is finished using that statement handle.
```

Figure 54. Example of improper serialization

Figure 55 shows a design that establishes a pool of connections. From this connection pool, you can map a Language Environment thread to each connection. This design prevents two Language Environment threads from using the same connection (or an associated statement handle) at the same time, but it allows these threads to share resources.



Parent Language Environment thread
SQLAllocHandle() (environment)
pthread_create **1**
pthread_create **2**
pthread_create ***m***
SQLAllocHandle() **1**
SQLAllocHandle() **2**
SQLAllocHandle() ***n*** (connection)
Connection pool
hdbc **1**
hdbc **2**
hdbc ***n***
Child Language Environment thread 1
SQLConnect() Connect to database A
SQLAllocHandle() (statement)
SQLExecDirect()
SQLEndTran()
SQLDisconnect()
Signal parent that hdbc is now available
Wait for more work
Mark a connection as in use
Pass handle to child thread
Return connection to pool
Mark connection free
When ready to shutdown:
SQLFreeHandle() **1 through *n***
Signal child threads to end
pthread_join **1 through *m***

Figure 55. Model for multithreading with connection pooling (MULTICONTEXT=1)

To establish a pool of connections (as Figure 55 on page 439 depicts), include the following steps in your application:

1. Designate a parent Language Environment thread. In DB2 ODBC, you designate a parent thread when you establish the environment with SQLAllocHandle(). This Language Environment thread that establishes the environment must persist for the duration of the application, so that DB2 language interface routines can remain resident in the Language Environment enclave.
2. From the parent Language Environment thread, allocate:
   - *m* child threads, one for each application task
   - *n* connection handles. This is the connection pool.
3. Execute each task on a separate child thread. Use the parent thread to dispatch these tasks to each child thread.
4. When a child thread requires access to a database, use the parent thread to allocate one of the n connections from the connection pool to the child thread. Remove this connection handle from the pool by marking it as used.
5. When you finish operating on a connection under a child thread, signal the parent thread to return this connection to the pool by marking it as free.
6. To terminate your application, free all connection handles with SQLFreeHandle() and terminate all child threads with pthread_join() from the parent thread.

Connections move from one application thread to another as the connections in the pool are assigned to child threads, returned to the pool, and assigned again.

With this design, you can create more Language Environment threads than connections, if threads are also used to perform non-SQL related tasks. You can also create more connections than threads, if you want to maintain a pool of active connections but limit the number of active tasks that your application performs.

DB2 ODBC does not control access to other application resources such as bound columns, parameter buffers, and files. If Language Environment threads need to share resources in your application, you must implement a mechanism to synchronize this access. Figure 52 on page 434 depicts an application that uses the Pthread functions to synchronize Language Environment threads that share a buffer.

External contexts

Typically, the DB2 ODBC driver manages contexts in an ODBC application. With external contexts, you can write applications that manage contexts outside of DB2 ODBC. You use external contexts in combination with Language Environment threads in the same way you use multiple contexts in combination with Language Environment threads. When you combine external contexts with Language Environment threads, you must manage both the external contexts and the Language Environment threads within your application.

To write an application that uses external contexts, specify the following values in the initialization file:

- MULTICONTEXT=0
- MVSATTACHTYPE=RRSAF

Call the following APIs in your application to manage contexts using Resource Recovery Services (RRS) instead of the DB2 ODBC driver:

- CRGGRM() to register your application as a resource manager
- CRGSEIF() to set exit routines for your application

- CTXBEGC() to create a private external context
- CTXSWCH() to switch between contexts
- CTXENDC() to end a private external context

When an application attempts to establish multiple active connections to the same data source from a single context, the ODBC driver rejects the connection request.

You cannot define different connection types for each external context. The following specifications set the connection type of all connections for every external context that your DB2 ODBC application creates:

- The CONNECTTYPE keyword in the initialization file
- The SQL_ATTR_CONNECTTYPE attribute in the functions SQLSetEnvAttr() and SQLSetConnectAttr()

See "CONNECTTYPE" on page 53 for more information about the CONNECTTYPE keyword.

DB2 ODBC does not support external contexts in applications that run as a stored procedure.

Because RRS is not part of DB2 ODBC, detailed information about RRS and RRS APIs is outside the scope of this book. For a complete description of RRS callable services, see *z/OS MVS Programming: Assembler Services Guide* or *z/OS MVS Programming: Resource Recovery*.

Figure 56 on page 442 shows an application that manages contexts outside of ODBC. This application uses RRS APIs to register as a context manager, set exit routines, create an external context, and switch between contexts.

```
/* Register as an unauthorized resource manager  */

CRGGRM();

/* Set exit information                          */

CRGSEIF();
/* Get an environment handle (henv)              */

SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

 /* Get a connection handle, hdbc1, and connect to
   STLEC1 under the native context.              */

SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc1);
SQLConnect( hdbc1, "STLEC1", ... );
/* Execute SQL under the native context at STLEC1*/

SQLAllocHandle(SQL_HANDLE_STMT, ...);
SQLExecDirect ...
     .
     .
/* Create a private context                      */

CTXBEGC( ctxtoken1 );

/* Switch to private                             */

CTXSWCH( ctxtoken1 );

An application that manages external contexts

/* Get a connection handle, hdbc2, and connect
   to STLEC1 under the private context.         */

SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc2);
SQLConnect( hdbc2, "STLEC1", ... );

/* Execute SQL under the private context at
   STLEC1                                         */

SQLAllocHandle(SQL_HANDLE_STMT, ...);
SQLExecDirect ...
     .
     .
/* Commit changes on hdbc2                       */

SQLEndTran(SQL_HANDLE_DBC,  hdbc2, SQL_COMMIT);
/* Switch back to native                         */

CTXSWCH( 0 );

/* Execute some more SQL under the native context
   at STLEC1                                      */

SQLAllocHandle(SQL_HANDLE_STMT, ...);
SQLExecDirect ...
    .
    .
/* Rollback changes on hdbc1                     */

SQLEndTran(SQL_HANDLE_DBC, hdbc1, SQL_ROLLBACK);
```

Figure 56. An application that manages external contexts

Application deadlocks

When you use multiple connections to access the same database resources concurrently, you create general contention for database resources. Timeouts and deadlocks can result from this contention.

The DB2 subsystem detects deadlocks and performs rollbacks on the necessary connections to resolve these deadlocks. However, the DB2 subsystem cannot detect a deadlock if the contention that created that deadlock involves application resources. An application that creates multiple connections with multithreading or multiple-context support can potentially create deadlocks if the following sequence occurs:

1. Two Language Environment threads connect to the same data source using two DB2 threads.
2. One Language Environment thread holds an internal application resource (such as a mutex) while its DB2 thread waits for access to a database resource.
3. The other Language Environment thread has a lock on a database resource while waiting for the internal application resource.

When this sequence of events occurs, the DB2 subsystem does not detect a deadlock because the DB2 subsystem cannot monitor the internal resources of the application. Although the DB2 subsystem cannot detect the deadlock itself, it does detect and handle any DB2 thread timeouts that result from that deadlock.

Handling application encoding schemes

This section describes DB2 ODBC support for EBCDIC, Unicode, and ASCII applications. Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The DB2 ODBC driver supports input and output character string arguments to ODBC Apes and input and output host variable data in each of these encoding schemes. With this support, you can manipulate data, SQL statements, and API string arguments in EBCDIC, Unicode, or ASCII.

Background

Different encoding schemes can represent character data. The EBCDIC and ASCII encoding scheme include multiple code pages; each code page represents 256 characters for one specific geography or one generic geography. The Unicode encoding scheme does not require the use of code pages, because it represents over 65 000 characters. Unicode can accommodate many different languages and geographies. Extensive information about the Unicode standard is available at www.unicode.org.

The Unicode standard defines several implementations including UTF-8, UCS-2, UTF-16, and UCS-4. ODBC DB2 supports Unicode in the following formats:
- UTF-8 (variable length, 1-byte to 6-byte characters)
- UCS-2 (2-byte characters)

Application programming guidelines

The DB2 ODBC driver determines whether an application is an EBCDIC, Unicode, or ASCII application by evaluating the setting of the CURRENTAPPENSCH keyword in the initialization file. You must compile your application with a compiler option that corresponds to this setting.

Specify corresponding encoding schemes for the DB2 ODBC driver and your application, by performing the following actions:

1. Set the CURRENTAPPENSCH keyword to EBCDIC, UNICODE, or ASCII. EBCDIC is the default.
2. Compile the application in EBCDIC, Unicode (with either the UTF-8 or UCS-2 compiler option), or ASCII.

You should specify the same encoding scheme with both of these actions.

When you write ODBC applications, you also need to choose API entry points and bind host variables to C types that are appropriate for the encoding scheme of your application.

Choosing an API entry point

A DB2 ODBC *entry point* is a function that provides support for one or more application encoding schemes. DB2 ODBC supports two entry points for each function that passes and accepts character string arguments: a generic API and a wide (suffix-W) API. The entry point that you use depends on the current encoding scheme of your application. Use the following guidelines to choose the correct entry points for your application:

- Use generic APIs for EBCDIC, ASCII, and Unicode UTF-8 string arguments.

  Example: To specify a Unicode UTF-8 argument, call a generic API:
  ```
  SQLExecDirect( (SQLHSTMT) hstmt,
                 (SQLCHAR *) UTF8STR,
                 (SQLINTEGER) SQL_NTS );
  ```
- Use wide (suffix-W) APIs only for Unicode UCS-2 string arguments. Wide APIs require that the CURRENTAPPENSCH keyword is set to UNICODE.

  Example: To specify a Unicode UCS-2 argument, call a suffix-W API:
  ```
  SQLExecDirectW( (SQLHSTMT) hstmt,
                 (SQLWCHAR *) UCS2STR,
                 (SQLINTEGER) SQL_NTS );
  ```

Table 230 on page 445 provides a detailed comparison of generic API and wide API syntax.

Binding host variables to C types

You use the generic APIs SQLBindCol(), SQLBindParameter(), and SQLGetData() as the entry points to bind application variables in all encoding schemes. DB2 ODBC requires only a single entry point to functions that bind application variables. The DB2 ODBC driver uses the following specifications to determine the encoding scheme of the character data in these functions:

- The *fCType* argument value in SQLBindCol(), SQLBindParameter(), and SQLGetData()
- The setting of the CURRENTAPPENSCH keyword in the DB2 ODBC initialization file

Table 229 summarizes how to set the CURRENTAPPENSCH keyword, declare application variables, and declare the *fCType* argument to bind application variables in each encoding scheme.

Table 229. Required values to bind application variables in each encoding scheme

DB2 ODBC elements	EBCDIC	Unicode UCS-2	Unicode UTF-8	ASCII
CURRENTAPPENSCH keyword setting	EBCDIC (default)	UNICODE	UNICODE	ASCII
Application variable C type definition	SQLCHAR or SQLDBCHAR	SQLWCHAR	SQLCHAR	SQLCHAR or SQLDBCHAR

Table 229. Required values to bind application variables in each encoding scheme (continued)

DB2 ODBC elements	EBCDIC	Unicode UCS-2	Unicode UTF-8	ASCII
fCType on SQLBindParameter(), SQLBindCol(), or SQLGetData()	SQL_C_CHAR or SQL_C_DBCHAR	SQL_C_WCHAR	SQL_C_CHAR	SQL_C_CHAR or SQL_C_DBCHAR

Requirement: You must use the symbolic C data type for the *fCType* argument that corresponds to the data type you use for application variables. For example, when you bind SQLCHAR application variables, you must specify the symbolic C data type SQL_C_CHAR for the *fCType* argument in your bind function call.

Suffix-W API function syntax

Table 230 compares the function prototypes for suffix-W APIs that DB2 UDB for z/OS supports with the function prototypes of their generic counterparts. The differences of the suffix-W function prototypes from the generic function prototypes are highlighted in **bold**.

Table 230. Comparison of suffix-W APIs to equivalent generic APIs

Generic APIs	Suffix-W APIs
SQLRETURN **SQLColAttributes** (SQLHSTMT hstmt, SQLUSMALLINT icol, SQLUSMALLINT fDescType, SQLPOINTER rgbDesc, SQLSMALLINT cbDescMax, SQLSMALLINT *pcbDesc, SQLINTEGER *pfDesc);	SQLRETURN **SQLColAttributesW** (SQLHSTMT hstmt, SQLUSMALLINT icol, SQLUSMALLINT fDescType, SQLPOINTER rgbDesc, SQLSMALLINT cbDescMax, SQLSMALLINT *pcbDesc, SQLINTEGER *pfDesc);
SQLRETURN **SQLColumns** (SQLHSTMT hstmt SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLCHAR *szColumnsName, SQLSMALLINT cbColumnName)	SQLRETURN **SQLColumnsW** (SQLHSTMT hstmt **SQLWCHAR** *szCatalogName, SQLSMALLINT cbCatalogName, **SQLWCHAR***szSchemaName, SQLSMALLINT cbSchemaName, **SQLWCHAR** *szTableName, SQLSMALLINT cbTableName, **SQLWCHAR** *szColumnsName, SQLSMALLINT cbColumnName);
SQLRETURN **SQLColumnPrivileges** (SQLHSTMT hstmt, SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLCHAR *szColumnsName, SQLSMALLINT cbColumnName);	SQLRETURN **SQLColumnPrivilegesW** (SQLHSTMT hstmt **SQLWCHAR** *szCatalogName, SQLSMALLINT cbCatalogName, **SQLWCHAR** *szSchemaName, SQLSMALLINT cbSchemaName, **SQLWCHAR** *szTableName, SQLSMALLINT cbTableName, **SQLWCHAR** *szColumnsName, SQLSMALLINT cbColumnName);
SQLRETURN **SQLConnect** (SQLHDBC hdbc, SQLCHAR *szDSN, SQLSMALLINT cbDSN, SQLCHAR *szUID, SQLSMALLINT cbUID, SQLCHAR *szAuthStr, SQLSMALLINT cbAuthStr);	SQLRETURN **SQLConnectW** (SQLHDBC hdbc, **SQLWCHAR** *szDSN, SQLSMALLINT cbDSN, **SQLWCHAR** *szUID, SQLSMALLINT cbUID, **SQLWCHAR** *szAuthStr, SQLSMALLINT cbAuthStr);

Table 230. Comparison of suffix-W APIs to equivalent generic APIs (continued)

Generic APIs	Suffix-W APIs
SQLRETURN **SQLDataSources** (	SQLRETURN **SQLDataSourcesW** (
SQLHENV henv,	SQLHENV henv,
SQLUSMALLINT fDirection,	SQLUSMALLINT fDirection,
SQLCHAR *szDSN,	**SQLWCHAR** *szDSN,
SQLSMALLINT cbDSNMax,	SQLSMALLINT cbDSNMax,
SQLSMALLINT *pcbDSN,	SQLSMALLINT *pcbDSN,
SQLCHAR *szDescription,	**SQLWCHAR** *szDescription,
SQLSMALLINT cbDescriptionMax,	SQLSMALLINT cbDescriptionMax,
SQLSMALLINT *pcbDescription);	SQLSMALLINT *pcbDescription);
SQLRETURN **SQLDescribeCol** (	SQLRETURN **SQLDescribeColW** (
SQLHSTMT hstmt,	SQLHSTMT hstmt,
SQLUSMALLINT icol,	SQLUSMALLINT icol,
SQLCHAR *szColName,	**SQLWCHAR** *szColName,
SQLSMALLINT cbColNameMax,	SQLSMALLINT cbColNameMax,
SQLSMALLINT *pcbColName,	SQLSMALLINT *pcbColName,
SQLSMALLINT *pfSqlType,	SQLSMALLINT *pfSqlType,
SQLUINTEGER *pcbColDef,	SQLUINTEGER *pcbColDef,
SQLSMALLINT *pibScale,	SQLSMALLINT *pibScale,
SQLSMALLINT *pfNullable);	SQLSMALLINT *pfNullable);
SQLRETURN **SQLDriverConnect** (	SQLRETURN **SQLDriverConnectW** (
SQLHDBC hdbc,	SQLHDBC hdbc,
SQLHWND hwnd,	SQLHWND hwnd,
SQLCHAR *szConnStrIn,	**SQLWCHAR** *szConnStrIn,
SQLSMALLINT cbConnStrIn,	SQLSMALLINT cbConnStrIn,
SQLCHAR *szConnStrOut,	**SQLWCHAR** *szConnStrOut,
SQLSMALLINT cbConnStrOutMax,	SQLSMALLINT cbConnStrOutMax,
SQLSMALLINT pcbConnStrOut,	SQLSMALLINT pcbConnStrOut,
SQLUSMALLINT fDriverCompletion);	SQLUSMALLINT fDriverCompletion);
SQLRETURN **SQLError** (	SQLRETURN **SQLErrorW** (
SQLHENV henv,	SQLHENV henv,
SQLHDBC hdbc,	SQLHDBC hdbc,
SQLHSTMT hstmt,	SQLHSTMT hstmt,
SQLCHAR *szSqlState,	**SQLWCHAR** *szSqlState,
SQLINTEGER *pfNativeError,	SQLINTEGER *pfNativeError,
SQLCHAR *szErrorMsg,	**SQLWCHAR** *szErrorMsg,
SQLSMALLINT cbErrorMsgMax,	SQLSMALLINT cbErrorMsgMax,
SQLSMALLINT *pcbErrorMsg);	SQLSMALLINT *pcbErrorMsg);
SQLRETURN **SQLExecDirect** (	SQLRETURN **SQLExecDirectW** (
SQLHSTMT hstmt,	SQLHSTMT hstmt,
SQLCHAR *szSqlStr,	**SQLWCHAR** *szSqlStr,
SQLINTEGER cbSqlStr);	SQLINTEGER cbSqlStr);
SQLRETURN **SQLForeignKeys** (	SQLRETURN **SQLForeignKeysW** (
SQLHSTMT hstmt,|	SQLHSTMT hstmt,|
SQLCHAR *szPkCatalogName,	**SQLWCHAR** *szPkCatalogName,
SQLSMALLINT :cbPkCatalogName,	SQLSMALLINT :cbPkCatalogName,
SQLCHAR *szPkSchemaName,	**SQLWCHAR** *szPkSchemaName,
SQLSMALLINT :cbPkSchemaName,	SQLSMALLINT :cbPkSchemaName,
SQLCHAR *szPkTableName,	**SQLWCHAR** *szPkTableName,
SQLSMALLINT :cbPkTableName,	SQLSMALLINT :cbPkTableName,
SQLCHAR *szFkCatalogName,	**SQLWCHAR** *szFkCatalogName,
SQLSMALLINT :cbFkCatalogName,	SQLSMALLINT :cbFkCatalogName,
SQLCHAR *szFkSchemaName,	**SQLWCHAR** *szFkSchemaName,
SQLSMALLINT :cbFkSchemaName,	SQLSMALLINT :cbFkSchemaName,
SQLCHAR *szFkTableName,	**SQLWCHAR** *szFkTableName,
SQLSMALLINT :cbFkTableName);	SQLSMALLINT :cbFkTableName);

Table 230. Comparison of suffix-W APIs to equivalent generic APIs (continued)

Generic APIs	Suffix-W APIs

SQLRETURN **SQLGetConnectOption** (
`SQLHDBC hdbc,`
`SQLUSMALLINT fOption,`
`SQLUINTEGER pvParam );`
	SQLRETURN **SQLGetConnectOptionW** (
	`SQLHDBC hdbc,`
	`SQLUSMALLINT fOption,`
	`SQLUINTEGER pvParam );`

SQLRETURN **SQLGetCursorName** (
`SQLHSTMT hstmt,`
`SQLCHAR *szCursor,`
`SQLSMALLINT cbCursorMax,`
`SQLSMALLINT *pcbCursor );`
	SQLRETURN **SQLGetCursorNameW** (
	`SQLHSTMT hstmt,`
	`SQLWCHAR` `*szCursor,`
	`SQLSMALLINT cbCursorMax,`
	`SQLSMALLINT *pcbCursor );`

SQLRETURN **SQLGetInfo** (
`SQLHDBC hdbc,`
`SQLUSMALLINT fInfoType,`
`SQLPOINTER rgbInfoValue,`
`SQLSMALLINT cbInfoValueMax,`
`SQLSMALLINT *pcbInfoValue );`
	SQLRETURN **SQLGetInfoW** (
	`SQLHDBC hdbc,`
	`SQLUSMALLINT fInfoType,`
	`SQLPOINTER rgbInfoValue,`
	`SQLSMALLINT cbInfoValueMax,`
	`SQLSMALLINT *pcbInfoValue );`

SQLRETURN **SQLGetStmtOption**(
`SQLHSTMT hstmt,`
`SQLUSMALLINT fOption,`
`SQLPOINTER pvParam );`
	SQLRETURN **SQLGetStmtOptionW**(
	`SQLHSTMT hstmt,`
	`SQLUSMALLINT fOption,`
	`SQLPOINTER pvParam );`

SQLRETURN **SQLGetTypeInfo** (
`SQLHSTMT hstmt,`
`SQLSMALLINT fSqlType );`
	SQLRETURN **SQLGetTypeInfoW** (
	`SQLHSTMT hstmt,`
	`SQLSMALLINT fSqlType );`

SQLRETURN **SQLNativeSql** (
`SQLHDBC hdbc,`
`SQLCHAR *szSqlStrIn,`
`SQLINTEGER cbSqlStrIn,`
`SQLCHAR *szSqlStr,`
`SQLINTEGER cbSqlStrMax,`
`SQLINTEGER *pcbSqlStr );`
	SQLRETURN **SQLNativeSqlW** (
	`SQLHDBC hdbc,`
	`SQLWCHAR` `*szSqlStrIn,`
	`SQLINTEGER cbSqlStrIn,`
	`SQLWCHAR` `*szSqlStr,`
	`SQLINTEGER cbSqlStrMax,`
	`SQLINTEGER *pcbSqlStr );`

SQLRETURN **SQLPrepare** (
`SQLHSTMT hstmt,`
`SQLCHAR *szSqlStr,`
`SQLINTEGER cbSqlStr );`
	SQLRETURN **SQLPrepareW** (
	`SQLHSTMT hstmt,`
	`SQLWCHAR` `*szSqlStr,`
	`SQLINTEGER cbSqlStr );`

SQLRETURN **SQLPrimaryKeys** (
`SQLHSTMT hstmt,`
`SQLCHAR *szCatalogName,`
`SQLSMALLINT :cbCatalogName,`
`SQLCHAR *szSchemaName,`
`SQLSMALLINT :cbSchemaName,`
`SQLCHAR *szTableName,`
`SQLSMALLINT :cbTableName );`
	SQLRETURN **SQLPrimaryKeysW** (
	`SQLHSTMT hstmt,`
	`SQLWCHAR` `*szCatalogName,`
	`SQLSMALLINT :cbCatalogName,`
	`SQLWCHAR` `*szSchemaName,`
	`SQLSMALLINT :cbSchemaName,`
	`SQLWCHAR` `*szTableName,`
	`SQLSMALLINT :cbTableName );`

SQLRETURN **SQLProcedureColumns** (
`SQLHSTMT hstmt,`
`SQLCHAR *szProcCatalog,`
`SQLSMALLINT cbProcCatalog,`
`SQLCHAR *szProcSchema,`
`SQLSMALLINT cbProcSchema,`
`SQLCHAR *szProcName,`
`SQLSMALLINT cbProcName,`
`SQLCHAR *szColumnName,`
`SQLSMALLINT cbColumnName );`
	SQLRETURN **SQLProcedureColumnsW** (
	`SQLHSTMT hstmt,`
	`SQLWCHAR` `*szProcCatalog,`
	`SQLSMALLINT cbProcCatalog,`
	`SQLWCHAR` `*szProcSchema,`
	`SQLSMALLINT cbProcSchema,`
	`SQLWCHAR` `*szProcName,`
	`SQLSMALLINT cbProcName,`
	`SQLWCHAR` `*szColumnName,`
	`SQLSMALLINT cbColumnName );`

Table 230. Comparison of suffix-W APIs to equivalent generic APIs (continued)

Generic APIs	Suffix-W APIs
SQLRETURN **SQLProcedures** (SQLHSTMT hstmt, SQLCHAR *szProcCatalog, SQLSMALLINT cbProcCatalog, SQLCHAR *szProcSchema, SQLSMALLINT cbProcSchema, SQLCHAR *szProcName, SQLSMALLINT cbProcName);	SQLRETURN **SQLProceduresW** (SQLHSTMT hstmt, **SQLWCHAR** *szProcCatalog, SQLSMALLINT cbProcCatalog, **SQLWCHAR** *szProcSchema, SQLSMALLINT cbProcSchema, **SQLWCHAR** *szProcName, SQLSMALLINT cbProcName);
SQLRETURN **SQLSetConnectOption** (SQLHDBC hdbc, SQLUSMALLINT fOption, SQLPOINTER pvParam);	SQLRETURN **SQLSetConnectOptionW** (SQLHDBC hdbc, SQLUSMALLINT fOption, SQLPOINTER pvParam);
SQLRETURN **SQLSetCursorName** (SQLHSTMT hstmt, SQLCHAR *szCursor, SQLSMALLINT cbCursor);	SQLRETURN **SQLSetCursorNameW** (SQLHSTMT hstmt, **SQLWCHAR** *szCursor, SQLSMALLINT cbCursor);
SQLRETURN **SQLSetStmtOption** (SQLHSTMT hstmt, SQLUSMALLINT fOption SQLUINTEGER pvParam);	SQLRETURN **SQLSetStmtOptionW** (SQLHSTMT hstmt, SQLUSMALLINT fOption SQLUINTEGER pvParam);
SQLRETURN **SQLSpecialColumns** (SQLHSTMT hstmt SQLUSMALLINT fColType, SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLUSMALLINT fScope, SQLUSMALLINT fNullable);	SQLRETURN **SQLSpecialColumnsW** (SQLHSTMT hstmt SQLUSMALLINT fColType, **SQLWCHAR** *szCatalogName, SQLSMALLINT cbCatalogName, **SQLWCHAR** *szSchemaName, SQLSMALLINT cbSchemaName, **SQLWCHAR** *szTableName, SQLSMALLINT cbTableName, SQLUSMALLINT fScope, SQLUSMALLINT fNullable);
SQLRETURN **SQLStatistics**(SQLHSTMT hstmt SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLUSMALLINT fUnique, SQLUSMALLINT fAccuracy);	SQLRETURN **SQLStatisticsW** (SQLHSTMT hstmt **SQLWCHAR** *szCatalogName, SQLSMALLINT cbCatalogName, **SQLWCHAR** *szSchemaName, SQLSMALLINT cbSchemaName, **SQLWCHAR** *szTableName, SQLSMALLINT cbTableName, SQLUSMALLINT fUnique, SQLUSMALLINT fAccuracy);
SQLRETURN **SQLTablePrivileges** (SQLHSTMT hstmt SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName);	SQLRETURN **SQLTablePrivilegesW** (SQLHSTMT hstmt **SQLWCHAR** *szCatalogName, SQLSMALLINT cbCatalogName, **SQLWCHAR** *szSchemaName, SQLSMALLINT cbSchemaName, **SQLWCHAR** *szTableName, SQLSMALLINT cbTableName);

Table 230. Comparison of suffix-W APIs to equivalent generic APIs (continued)

Generic APIs	Suffix-W APIs
SQLRETURN **SQLTables** (	SQLRETURN **SQLTablesW** (
SQLHSTMT hstmt SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLCHAR *szTableType, SQLSMALLINT cbTableType);	SQLHSTMT hstmt **SQLWCHAR** *szCatalogName, SQLSMALLINT cbCatalogName, **SQLWCHAR** *szSchemaName, SQLSMALLINT cbSchemaName, **SQLWCHAR** *szTableName, SQLSMALLINT cbTableName, **SQLWCHAR** *szTableType, SQLSMALLINT cbTableType);

Examples of handling the application encoding scheme

The following examples demonstrate how to declare variables, specify data types, and use suffix-W APIs appropriately for a particular encoding scheme. These examples demonstrate binding a UCS-2 result set column, binding UTF-8 data to parameter markers, retrieving UTF-8 data into application variables, and using suffix-W APIs.

Binding result set columns to retrieve UCS-2 data

The following code uses SQLBindCol() to bind the first column of a result set to a Unicode UCS-2 application buffer.

```
/* Declare variable to bind Unicode UCS-2 data */

SQLWCHAR     UCSWSTR   [50];

/* Assume CURRENTAPPENSCH=UNICODE is set */

SQLBindCol( (SQLHSTMT) hstmt,
            (SQLUSMALLINT) 1,
            (SQLSMALLINT) SQL_C_WCHAR,
            (SQLPOINTER) UCSWSTR,
            (SQLINTEGER) sizeof(UCSWSTR),
            (SQLINTEGER*)&amp;LEN_UCSWSTR);
```

Binding UTF-8 data to parameter markers

In Figure 57 on page 450, SQLBindParameter() binds three input application variables that contain UTF-8 data to INTEGER, CHAR, and GRAPHIC parameter markers.

```
/* Declare variables for Unicode UTF-8 data */
SQLCHAR     HV1INT    [50];
SQLCHAR     HV1CHAR   [50];
SQLCHAR     HV1GRAPHIC[50];

SQLINTEGER  LEN_HV1INT;
SQLINTEGER  LEN_HV1CHAR;
SQLINTEGER  LEN_HV1GRAPHIC;

...

/* Assume CURRENTAPPENSCH=UNICODE is set */

/* Bind to DB2 INTEGER */

SQLBindParameter( (SQLHSTMT)     hstmt,
                  (SQLUSMALLINT) 1,
                  (SQLSMALLINT)  SQL_PARAM_INPUT,
                  (SQLSMALLINT)  SQL_C_CHAR,
                  (SQLSMALLINT)  SQL_INTEGER,
                  (SQLUINTEGER)  0,
                  (SQLSMALLINT)  0,
                  (SQLPOINTER)   HV1INT,
                  (SQLINTEGER)   sizeof(HV1INT),
                  (SQLINTEGER *) &LEN_HV1INT ),

/* Bind to DB2 CHAR(10) */

SQLBindParameter( (SQLHSTMT)     hstmt,
                  (SQLUSMALLINT) 2,
                  (SQLSMALLINT)  SQL_PARAM_INPUT,
                  (SQLSMALLINT)  SQL_C_CHAR,
                  (SQLSMALLINT)  SQL_CHAR,
                  (SQLUINTEGER)  10,
                  (SQLSMALLINT)  0,
                  (SQLPOINTER)   HV1CHAR,
                  (SQLINTEGER)   sizeof(HV1CHAR),
                  (SQLINTEGER *) &LEN_HV1CHAR ),

/* Bind to DB2 GRAPHIC(20) */

SQLBindParameter( (SQLHSTMT)     hstmt,
                  (SQLUSMALLINT) 3,
                  (SQLSMALLINT)  SQL_PARAM_INPUT,
                  (SQLSMALLINT)  SQL_C_CHAR,
                  (SQLSMALLINT)  SQL_GRAPHIC,
                  (SQLUINTEGER)  20,
                  (SQLSMALLINT)  0,
                  (SQLPOINTER)   HV1GRAPHIC,
                  (SQLINTEGER)   sizeof(HV1GRAPHIC),
                  (SQLINTEGER *) &LEN_HV1GRAPHIC );
```

Figure 57. An application that binds application variables to parameter markers

Retrieving UTF-8 data into application variables

In Figure 58 on page 451, SQLGetData() retrieves UTF-8 data from three columns (DECIMAL, VARCHAR, and VARGRAPHIC) in the current row of the result set.

```
/* Declare variables for Unicode UTF-8 data */

SQLCHAR     HV1DECIMAL   [50];
SQLCHAR     HV1VARCHAR   [100];
SQLCHAR     HV1VARGRAPHIC[200];

SQLINTEGER  LEN_HV1DECIMAL;
SQLINTEGER  LEN_HV1VARCHAR;
SQLINTEGER  LEN_HV1VARGRAPHIC;

...


/* Assume CURRENTAPPENSCH=UNICODE is set */

/* Bind DECIMAL(10,2) column */

SQLGetData( (SQLHSTMT)     hstmt,
            (SQLUSMALLINT) 1,
            (SQLSMALLINT)  SQL_C_CHAR,
            (SQLPOINTER)   HV1DECIMAL,
            (SQLINTEGER)   sizeof(HV1DECIMAL),
            (SQLINTEGER *) &LEN_HV1DECIMAL ),

/* Bind VARCHAR(20) column */

SQLGetData( (SQLHSTMT)     hstmt,
            (SQLUSMALLINT) 2,
            (SQLSMALLINT)  SQL_C_CHAR,
            (SQLPOINTER)   HV1VARCHAR,
            (SQLINTEGER)   sizeof(HV1VARCHAR),
            (SQLINTEGER *) &LEN_HV1VARCHAR ),


/* Bind VARGRAPHIC(30) column */

SQLGetData( (SQLHSTMT)     hstmt,
            (SQLUSMALLINT) 3,
            (SQLSMALLINT)  SQL_C_CHAR,
            (SQLPOINTER)   HV1VARGRAPHIC,
            (SQLINTEGER)   sizeof(HV1VARGRAPHIC),
            (SQLINTEGER *) &LEN_HV1VARGRAPHIC );
```

Figure 58. An application that retrieves result set data into application variables

Using suffix-W APIs

Figure 59 on page 452 shows an example ODBC application that uses three suffix-W APIs to handle a Unicode UCS-2 application encoding scheme.

```
/******************************************************************/
/*  Main program                                                  */
/*  - CREATE MYTABLE                                              */
/*  - INSERT INTO MYTABLE using literals                          */
/*  - INSERT INTO MYTABLE using parameter markers                 */
/*  - SELECT FROM MYTABLE with WHERE clause                       */
/*                                                                */
/*  suffix-W APIS used:                                           */
/*  - SQLConnectW                                                 */
/*  - SQLPrepareW                                                 */
/*  - SQLExecDirectW                                              */
/******************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <wstr.h>
#include "sqlcli1.h"
#include <stlca.h>
#include <errno.h>
#include <sys/_messag.h>

#pragma convlit(suspend)

    SQLHENV         henv  = SQL_NULL_HENV;
    SQLHDBC         hdbc  = SQL_NULL_HDBC;
    SQLHSTMT        hstmt = SQL_NULL_HSTMT;
    SQLRETURN       rc    = SQL_SUCCESS;
    SQLINTEGER      id;
    SQLSMALLINT     scale;
    SQLCHAR         server[18]
    SQLCHAR         uid[30]
    SQLCHAR         pwd[30]
    SQLSMALLINT     pcpar=0;
    SQLSMALLINT     pccol=0;
    SQLCHAR         sqlstmt[200]
    SQLINTEGER      sqlstmtlen;
    SQLWCHAR        H1INT4      [50]
    SQLWCHAR        H1SMINT     [50]
    SQLWCHAR        H1CHR10     [50]
    SQLWCHAR        H1CHR10MIX  [50]
    SQLWCHAR        H1VCHR20    [50]
    SQLWCHAR        H1VCHR20MIX [50]
    SQLWCHAR        H1GRA10     [50]
    SQLWCHAR        H1VGRA20    [50]
    SQLWCHAR        H1TTIME     [50]
    SQLWCHAR        H1DDATE     [50]
    SQLWCHAR        H1TSTMP     [50]
    SQLWCHAR        H2INT4      [50]
```

Figure 59. An application that uses suffix-W APIs (Part 1 of 12)

```
SQLWCHAR        H2SMINT     [50]
    SQLWCHAR        H2CHR10     [50]
    SQLWCHAR        H2CHR10MIX  [50]
    SQLWCHAR        H2VCHR20    [50]
    SQLWCHAR        H2VCHR20MIX [50]
    SQLWCHAR        H2GRA10     [50]
    SQLWCHAR        H2VGRA20    [50]
    SQLWCHAR        H2TTIME     [50]
    SQLWCHAR        H2DDATE     [50]
    SQLWCHAR        H2TSTMP     [50]

    SQLINTEGER      LEN_H1INT4;
    SQLINTEGER      LEN_H1SMINT;
    SQLINTEGER      LEN_H1CHR10;
    SQLINTEGER      LEN_H1CHR10MIX;
    SQLINTEGER      LEN_H1VCHR20;
    SQLINTEGER      LEN_H1VCHR20MIX;
    SQLINTEGER      LEN_H1GRA10;
    SQLINTEGER      LEN_H1VGRA20;
    SQLINTEGER      LEN_H1TTIME;
    SQLINTEGER      LEN_H1DDATE;
    SQLINTEGER      LEN_H1TSTMP;
    SQLINTEGER      LEN_H2INT4;
    SQLINTEGER      LEN_H2SMINT;
    SQLINTEGER      LEN_H2CHR10;
    SQLINTEGER      LEN_H2CHR10MIX;
    SQLINTEGER      LEN_H2VCHR20;
    SQLINTEGER      LEN_H2VCHR20MIX;
    SQLINTEGER      LEN_H2GRA10;
    SQLINTEGER      LEN_H2VGRA20;
    SQLINTEGER      LEN_H2TTIME;
    SQLINTEGER      LEN_H2DDATE;
    SQLINTEGER      LEN_H2TSTMP;
    SQLWCHAR        DROPW1  [100]
    SQLWCHAR        DELETEW1[100]
    SQLWCHAR        SELECTW1[100]
    SQLWCHAR        CREATEW1[500]
    SQLWCHAR        INSERTW1[500]

    SQLWCHAR        DROPW2  [100]
    SQLWCHAR        DELETEW2[100]
    SQLWCHAR        SELECTW2[100]
    SQLWCHAR        CREATEW2[500]
    SQLWCHAR        INSERTW2[500]

    SQLINTEGER      LEN_H1INT4;
    SQLINTEGER      LEN_DROPW1;
    SQLINTEGER      LEN_DELETEW1;
    SQLINTEGER      LEN_INSERTW1;
    SQLINTEGER      LEN_CREATEW1;
    SQLINTEGER      LEN_SELECTW1;
```

Figure 59. An application that uses suffix-W APIs (Part 2 of 12)

```
    SQLINTEGER      LEN_DROPW2;
    SQLINTEGER      LEN_DELETEW2;
    SQLINTEGER      LEN_INSERTW2;
    SQLINTEGER      LEN_CREATEW2;
    SQLINTEGER      LEN_SELECTW2;

    struct {
      short LEN;
      char  DATA&lbracket;200&rbracket;; }  STMTSQL;
    long               SPCODE;
    int                result;
    int                ix, locix;
/******************************************************************/
int main()
{
  henv=0;
  rc=SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
  if( rc != SQL_SUCCESS ) goto dberror;
  hdbc=0;
  rc=SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
  if( rc != SQL_SUCCESS ) goto dberror;
  /****************************************************************/
  /*  Setup application host variables (UCS-2 character strings)   */
  /****************************************************************/
#pragma convlit(resume)
  wcscpy(uid, (wchar_t *)"jgold");
  wcscpy(pwd, (wchar_t *)"general");
  wcscpy(server, (wchar_t *)"STLEC1");
  wcscpy(DROPW1, (wchar_t *)
    "DROP TABLE MYTABLE");
  LEN_DROPW1=wcslen((wchar_t *)DROPW1);
  wcscpy(SELECTW1, (wchar_t *)
    "SELECT * FROM MYTABLE WHERE INT4=200");
  LEN_SELECTW1=wcslen((wchar_t *)SELECTW1);
  wcscpy(CREATEW1, (wchar_t *)
    "CREATE TABLE MYTABLE ( ");
  wcscat(CREATEW1, (wchar_t *)
    "INT4 INTEGER, SMINT SMALLINT, ");
  wcscat(CREATEW1, (wchar_t *)
    "CHR10 CHAR(10), CHR10MIX CHAR(10) FOR MIXED DATA, ");
```

Figure 59. An application that uses suffix-W APIs (Part 3 of 12)

```
  wcscat(CREATEW1, (wchar_t *)
    "VCHR20 VARCHAR(20), VCHR20MIX VARCHAR(20) FOR MIXED DATA, ");
  wcscat(CREATEW1, (wchar_t *)
    "GRA10 GRAPHIC(10), VGRA20 VARGRAPHIC(20), ");
  wcscat(CREATEW1, (wchar_t *)
    "TTIME TIME, DDATE DATE, TSTMP TIMESTAMP )" );
  LEN_CREATEW1=wcslen((wchar_t *)CREATEW1);
  wcscpy(DELETEW1, (wchar_t *)
    "DELETE FROM MYTABLE WHERE INT4 IS NULL OR INT4 IS NOT NULL");
  LEN_DELETEW1=wcslen((wchar_t *)DELETEW1);
  wcscpy(INSERTW1, (wchar_t *)
    "INSERT INTO MYTABLE VALUES ( ");
  wcscat(INSERTW1, (wchar_t *)
    "( 100,1,'CHAR10','CHAR10MIX','VARCHAR20','VARCHAR20MIX', ");
  wcscat(INSERTW1, (wchar_t *)
    "G' A B C', VARGRAPHIC('ABC'), ");
  wcscat(INSERTW1, (wchar_t *)
    "'3:45 PM', '06/12/1999', ");
  wcscat(INSERTW1, (wchar_t *)
    "'1999-09-09-09.09.09.090909' )" );
  LEN_INSERTW1=wcslen((wchar_t *)INSERTW1);
  wcscpy(INSERTW2, (wchar_t *)
    "INSERT INTO MYTABLE VALUES (?,?,?,?,?,?,?,?,?,?,?)");
  LEN_INSERTW2=wcslen((wchar_t *)INSERTW2);
  wcscpy(H1INT4     , (wchar_t *)"200");
  wcscpy(H1SMINT    , (wchar_t *)"5");
  wcscpy(H1CHR10    , (wchar_t *)"CHAR10");
  wcscpy(H1CHR10MIX , (wchar_t *)"CHAR10MIX");
  wcscpy(H1VCHR20   , (wchar_t *)"VARCHAR20");
  wcscpy(H1VCHR20MIX, (wchar_t *)"VARCHAR20MIX");
  wcscpy(H1TTIME    , (wchar_t *)"3:45 PM");
  wcscpy(H1DDATE    , (wchar_t *)"06/12/1999");
  wcscpy(H1TSTMP    , (wchar_t *)"1999-09-09-09.09.09.090909");
#pragma convlit(suspend)
  /* 0xFF21,0xFF22,0xFF23,0x0000 */
  wcscpy(H1GRA10    , (wchar_t *)"    ");
  /* 0x0041,0xFF21,0x0000 */
  wcscpy(H1VGRA20   , (wchar_t *)"    ");
  LEN_H1INT4      = SQL_NTS;
  LEN_H1SMINT = SQL_NTS;
  LEN_H1CHR10     = SQL_NTS;
  LEN_H1CHR10MIX  = SQL_NTS;
  LEN_H1VCHR20    = SQL_NTS;
  LEN_H1VCHR20MIX = SQL_NTS;
  LEN_H1GRA10     = SQL_NTS;
  LEN_H1VGRA20    = SQL_NTS;
  LEN_H1TTIME     = SQL_NTS;
  LEN_H1DDATE     = SQL_NTS;
  LEN_H1TSTMP     = SQL_NTS;
```

Figure 59. An application that uses suffix-W APIs (Part 4 of 12)

```
/*******************************************************************/
/*  SQLConnectW                                                    */
/*******************************************************************/
rc=SQLConnectW(hdbc, NULL, 0, NULL, 0, NULL, 0);
if( rc != SQL_SUCCESS ) goto dberror;
/*******************************************************************/
/*  DROP TABLE - SQLExecuteDirectW                                 */
/*******************************************************************/
hstmt=0;
rc=SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
rc=SQLExecDirectW(hstmt,DROPW1,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;
rc=SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
/*******************************************************************/
/*  CREATE TABLE MYTABLE - SQLPrepareW                             */
/*******************************************************************/
hstmt=0;
rc=SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
rc=SQLPrepareW(hstmt,CREATEW1,SQL_NTS);
if( rc != SQL_SUCCESS) goto dberror;
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;
rc=SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
/*******************************************************************/
/*  INSERT INTO MYTABLE with literals - SQLExecDirectW             */
/*******************************************************************/
hstmt=0;
rc=SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
rc=SQLExecDirectW(hstmt,DROPW1,SQL_NTS);
if( rc != SQL_SUCCESS) goto dberror;
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS) goto dberror;
rc=SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
if( rc != SQL_SUCCESS) goto dberror;
```

Figure 59. An application that uses suffix-W APIs (Part 5 of 12)

```
  /*****************************************************************/
  /*  INSERT INTO MYTABLE with parameter markers                   */
  /*  - SQLPrepareW                                                */
  /*  - SQLBindParameter with SQL_C_WCHAR symbolic C data type     */
  /*****************************************************************/
  hstmt=0;
      rc=SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
  if( rc != SQL_SUCCESS ) goto dberror;
  /*  INSERT INTO MYTABLE VALUES (?,?,?,?,?,?,?,?,?,?,?)  */
  rc=SQLPrepareW(hstmt,INSERTW2,SQL_NTS);
  if( rc != SQL_SUCCESS ) goto dberror;
  rc=SQLNumParams(hstmt, &pcpar);
  if( rc != SQL_SUCCESS) goto dberror;
  printf("\nAPDV1                                 number= 19");
  if( pcpar != 11 ) goto dberror;
  /* Bind INTEGER parameter */
  rc= SQLBindParameter(hstmt,
                      1,
                      SQL_PARAM_INPUT,
                      SQL_C_WCHAR,
                      SQL_INTEGER,
                      10,
                      0,
                      (SQLPOINTER)H1INT4,
                      sizeof(H1INT4
),                    (SQLINTEGER *)&LEN_H1INT4 );
  if( rc != SQL_SUCCESS) goto dberror;
/* Bind SMALLINT parameter */
rc = SQLBindParameter(hstmt,
                      2,
                      SQL_PARAM_INPUT,
                      SQL_C_WCHAR,
                      SQL_SMALLINT,
                      5,
                      0,
                      (SQLPOINTER)H1SMINT,
                      sizeof(H1SMINT),
                      (SQLINTEGER*)&LEN_H1SMINT);
 if( rc != SQL_SUCCESS ) goto dberror;

  /* Bind CHAR(10) parameter */
  rc = SQLBindParameter(hstmt,
                       3,
                       SQL_PARAM_INPUT,
                       SQL_C_WCHAR,
                       SQL_CHAR,
                       10,
```

Figure 59. An application that uses suffix-W APIs (Part 6 of 12)

```
                0,
                      (SQLPOINTER)H1CHR10,
                      sizeof(H1CHR10),
                      (SQLINTEGER *)&LEN_H1CHR10);
 if( rc != SQL_SUCCESS ) goto dberror;
 /* Bind CHAR(10) parameter */
 rc = SQLBindParameter(hstmt,
                      3,
                      SQL_PARAM_INPUT,
                      SQL_C_WCHAR,
                      SQL_CHAR,
                      10,
                      0,
                      (SQLPOINTER)H1CHR10,
                      sizeof(H1CHR10),
                      (SQLINTEGER *)&LEN_H1CHR10);
 if( rc != SQL_SUCCESS ) goto dberror;
 /* Bind CHAR(10) FOR MIXED parameter */
 rc = SQLBindParameter(hstmt,
                       4,
                      SQL_PARAM_INPUT,
                      SQL_C_WCHAR,
                      SQL_CHAR,
                      10,
                      0,
                      (SQLPOINTER)H1CHR10MIX,
                      sizeof(H1CHR10MIX),
                      (SQLINTEGER *)&LEN_H1CHR10MIX);
 if( rc != SQL_SUCCESS ) goto dberror;
/* Bind VARCHAR(20) parameter */
rc = SQLBindParameter(hstmt,
                       5,
                       SQL_PARAM_INPUT,
                       SQL_C_WCHAR,
                       SQL_VARCHAR,
                       20,
                       0,
                       (SQLPOINTER)H1VCHR20,
                       sizeof(H1VCHR20),
                       (SQLINTEGER *)&LEN_H1VCHR20);
 if( rc != SQL_SUCCESS ) goto dberror;
 /* Bind VARCHAR(20) FOR MIXED parameter */
 rc = SQLBindParameter(hstmt,
                      6,
                      SQL_PARAM_INPUT,
                      SQL_C_WCHAR,
                      SQL_VARCHAR,
```

Figure 59. An application that uses suffix-W APIs (Part 7 of 12)

```
                      20,
                      0,
                      (SQLPOINTER)H1VCHR20MIX,
                      sizeof(H1VCHR20MIX),
                      (SQLINTEGER *)&LEN_H1VCHR20MIX);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind GRAPHIC(10) parameter */

rc = SQLBindParameter(hstmt,
                      7,
                      SQL_PARAM_INPUT,
                      SQL_C_WCHAR,
                      SQL_GRAPHIC,
                      10,
                      0,
                      (SQLPOINTER)H1GRA10,
                      sizeof(H1GRA10),
                      (SQLINTEGER *)&LEN_H1GRA10);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind VARGRAPHIC(20) parameter*/
rc = SQLBindParameter(hstmt,
                      8,
                      SQL_PARAM_INPUT,
                      SQL_C_WCHAR,
                      SQL_VARGRAPHIC,
                      20,
                      0,
                      (SQLPOINTER)H1VGRA20,
                      sizeof(H1VGRA20),
                      (SQLINTEGER *)&LEN_H1VGRA20);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind TIME parameter */
rc= SQLBindParameter(hstmt,
                     9,
                     SQL_PARAM_INPUT,
                     SQL_C_WCHAR,
                     SQL_TIME,
                     8,
                     0,
                     (SQLPOINTER)H1TTIME,
                     sizeof(H1TTIME),
                     (SQLINTEGER *)&LEN_H1TTIME);
if( rc != SQL_SUCCESS) goto dberror;
/* Bind DATE parameter */
rc = SQLBindParameter(hstmt,
                      10,
                      SQL_PARAM_INPUT,
```

Figure 59. An application that uses suffix-W APIs (Part 8 of 12)

```
                      SQL_C_WCHAR,
                      SQL_DATE,
                      10,
                      0,
                      (SQLPOINTER)H1DDATE,
                      sizeof(H1DDATE),
                      (SQLINTEGER *)&LEN_H1DDATE);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind TIMESTAMP parameter */
rc = SQLBindParameter(hstmt,
                      11
                      SQL_PARAM_INPUT,
                      SQL_C_WCHAR,
                      SQL_DATE,
                      26,
                      0,
                      (SQLPOINTER)H1TSTMP,
                      sizeof(H1TSTMP),
                      (SQLINTEGER *)&LEN_H1TSTMP);
if( rc != SQL_SUCCESS ) goto dberror;
printf("\nAPDV1 SQLExecute                     number= 25");
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS) goto dberror;
printf("\nAPDV1 SQLEndTran                     number=26");
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;
printf("\nAPDV1 SQLFreeHandle(SQL_HANDLE_STMT, ...) number= 27");
rc=SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
if( rc != SQL_SUCCESS ) goto dberror;

/*****************************************************************/
/*  SELECT FROM MYTABLE WHERE INT4=200                           */
/*  - SQLBindCol with SQL_C_WCHAR symbolic C data type           */
/*  - SQLExecDirectW                                             */
/*****************************************************************/
hstmt=0;
rc=SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind INTEGER column */

rc = SQLBindCol(hstmt,
                1,
                SQL_C_WCHAR,
                (SQLPOINTER)H2INT4,
                sizeof(H2INT4 ),
                (SQLINTEGER *)&LEN_H2INT4 );
if( rc != SQL_SUCCESS ) goto dberror;
```

Figure 59. An application that uses suffix-W APIs (Part 9 of 12)

```
/* Bind SMALLINT column */
rc = SQLBindCol(hstmt,
                2,
                SQL_C_WCHAR,
                (SQLPOINTER)H2SMINT,
                sizeof(H2SMINT),
                (SQLINTEGER *)&LEN_H2SMINT);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind CHAR(10) column */
rc = SQLBindCol(hstmt,
                3,
                SQL_C_WCHAR,
                (SQLPOINTER)H2CHR10,
                sizeof(H2CHR10),
                (SQLINTEGER *)&LEN_H2CHR10);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind CHAR(10) FOR MIXED column */
rc = SQLBindCol(hstmt,
                4,
                SQL_C_WCHAR,
                (SQLPOINTER)H2CHR10MIX,
                sizeof(H2CHR10MIX),
                (SQLINTEGER *)&LEN_H2CHR10MIX);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind VARCHAR(20) column */
rc = SQLBindCol(hstmt,
                5,
                SQL_C_WCHAR,
                (SQLPOINTER)H2VCHR20,
                sizeof(H2VCHR20,
                (SQLINTEGER *)&LEN_H2VCHR20);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind VARCHAR(20) FOR MIXED column */
rc = SQLBindCol(hstmt,
                6,
                SQL_C_WCHAR,
                (SQLPOINTER)H2VCHR20MIX,
                sizeof(H2VCHR20MIX),
                (SQLINTEGER *)&LEN_H2VCHR20MIX);
if( rc != SQL_SUCCESS ) goto dberror;
```

Figure 59. An application that uses suffix-W APIs (Part 10 of 12)

```
/* Bind GRAPHIC(10) column */
rc = SQLBindCol(hstmt,
                7,
                SQL_C_WCHAR,
                (SQLPOINTER)H2GRA10,
                sizeof(H2GRA10),
                (SQLINTEGER *)&LEN_H2GRA10);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind VARGRAPHIC(20) column */
rc = SQLBindCol(hstmt,
                8,
                SQL_C_WCHAR,
                (SQLPOINTER)H2VGRA20,
                sizeof(H2VGRA20),
                (SQLINTEGER *)&LEN_H2VGRA20);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind TIME column */
rc = SQLBindCol(hstmt,
                9,
                SQL_C_WCHAR,
                (SQLPOINTER)H2TTIME,
                sizeof(H2TTIME),
                (SQLINTEGER *)&LEN_H2TTIME);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind DATE column */
rc = SQLBindCol(hstmt,
                10,
                SQL_C_WCHAR,
                (SQLPOINTER)H2DDATE,
                sizeof(H2DDATE),
                (SQLINTEGER *)&LEN_H2DDATE);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind TIMESTAMP column */
rc = SQLBindCol(hstmt,
                11,
                SQL_C_WCHAR,
                (SQLPOINTER)H2TSTMP,
                sizeof(H2TSTMP),
                (SQLINTEGER *)&LEN_H2TSTMP);
if( rc != SQL_SUCCESS ) goto dberror;
/*
 *   SELECT * FROM MYTABLE WHERE INT4=200
 */
```

Figure 59. An application that uses suffix-W APIs (Part 11 of 12)

```
  rc=SQLExecDirectW(hstmt,SELECTW1,SQL_NTS);
  if( rc != SQL_SUCCESS ) goto dberror;
  rc=SQLFetch(hstmt);
  if( rc != SQL_SUCCESS ) goto dberror;
  rc=SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
  if( rc != SQL_SUCCESS ) goto dberror;

/*****************************************************************/
  rc=SQLDisconnect(hdbc);
  if( rc != SQL_SUCCESS ) goto dberror;
  rc=SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
  if( rc != SQL_SUCCESS ) goto dberror;
  rc=SQLFreeHandle(SQL_HANDLE_ENV, henv);
  if( rc != SQL_SUCCESS ) goto dberror;
  dberror:
  rc = SQL_ERROR;
  return(rc);

}  /*END MAIN*/
```

Figure 59. An application that uses suffix-W APIs (Part 12 of 12)

Mixing embedded SQL with DB2 ODBC

You can combine embedded static SQL with DB2 ODBC to write a mixed application. With a mixed application, you can take advantage of both the ease of use that DB2 ODBC functions provide and the performance enhancement that embedded SQL offers.

Important: To mix DB2 ODBC with embedded SQL, you must not enable DB2 ODBC support for multiple contexts. The initialization file for mixed applications must specify MULTICONTEXT=0 or exclude MULTICONTEXT keyword.

To mix DB2 ODBC and embedded SQL in an application, you must limit how you combine these interfaces:

- Handle all connection management and transaction management with either DB2 ODBC or embedded SQL exclusively. You must perform all connections, commits, and rollbacks with the same interface.
- Use only one interface (DB2 ODBC or embedded SQL) for each query statement. For example, an application cannot open a cursor in an embedded SQL routine, and then call the DB2 ODBC SQLFetch() function to retrieve row data.

Because DB2 ODBC permits multiple connections, you must call SQLSetConnection() before you call a routine that is written in embedded SQL. SQLSetConnection() allows you to explicitly specify the connection on which you want the embedded SQL routine to run. If your application makes only a single connection, or if you write your application entirely in DB2 ODBC, you do not need to include a SQLSetConnection() call.

Tip: When you write a mixed application, divide this application into a main program that makes separate function calls. Structure the mixed application as a DB2 ODBC application that calls functions that are written with embedded SQL, or as an embedded SQL application that calls functions that are written with DB2 ODBC. With this kind of structure, you can perform transaction management separately in the main program, while you make query statements in individual functions written in a single interface. Functions that are written with DB2 ODBC must use null connections. See “Writing a DB2 ODBC stored procedure” on page 430 for details about null connections.

Figure 60 shows an application that connects to two data sources and executes both embedded SQL and dynamic SQL using DB2 ODBC.

```
/* ... */
    /* Allocate an environment handle   */
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &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 two connections are active */

    /* 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=%ld\n", rc);

   /*  Commit transaction on connection 1 */
    SQLEndTran(SQL_HANDLE_DBC, 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=%ld\n", rc);

   /*  Commit transaction on connection 2 */
    SQLEndTran(SQL_HANDLE_DBC, 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=%ld\n", rc);
```

Figure 60. An application that mixes embedded and dynamic SQL (Part 1 of 2)

```
   /*  Commit transaction on connection 1 */
    SQLEndTran(SQL_HANDLE_DBC, hdbc[0], SQL_COMMIT);

    SQLSetConnection(hdbc[1]);
    if (( rc = Drop_Tab() ) != 0)
        printf("Error dropping Table on 2nd connection, RC=%ld\n", rc);

   /*  Commit transaction on connection 2 */
    SQLEndTran(SQL_HANDLE_DBC, hdbc[1], SQL_COMMIT);

    printf("Tables dropped\n");
    /*********   End processing step  ***************************/

/* ... */
/*************   Embedded SQL functions  *******************************
** This would normally be a separate file to avoid having to          *
** keep precompiling the embedded file in order to compile the DB2 CLI *
** section50                                                           *
***********************************************************************/

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);
}
/* ... */
```

Figure 60. An application that mixes embedded and dynamic SQL (Part 2 of 2)

Using vendor escape clauses

If your application accesses only DB2 data sources, you have no reason to use vendor escape clauses. If your application accesses multiple data sources from different vendors, vendor escape clauses increase the portability of your application.

The X/Open SQL CAE specification defines an *escape clause* as: “a syntactic mechanism for vendor-specific SQL extensions to be implemented in the framework of standardized SQL.” Both DB2 ODBC and ODBC support *vendor escape clauses* that conform to this X/Open specification.

Data sources are not necessarily consistent in how they implement SQL extensions. Use vendor escape clauses to implement common SQL extensions in a consistent, portable format. Common SQL extensions are listed in “ODBC-defined SQL extensions” on page 467.

DB2 ODBC translates the SQL extensions that ODBC defines to native DB2 SQL syntax. To display the DB2-specific syntax that results from this translation, call SQLNativeSql() on an SQL string that contains ODBC vendor escape clauses.

Determining ODBC vendor escape clause support

Data sources do not necessarily support the same SQL extensions. The ODBC drivers for these data sources therefore might not support all ODBC vendor escape clauses.

To determine if a data source supports vendor escape clauses, call SQLGetInfo() with the *InfoType* argument set to SQL_ODBC_SQL_CONFORMANCE. If SQLGetInfo returns a value of SQL_OSC_EXTENDED, that data source supports all ODBC vendor escape clauses.

For information about how to determine if a data source supports a specific ODBC vendor escape clause, refer to the descriptions of each individual escape clause in "ODBC-defined SQL extensions" on page 467.

For SQL extensions that ODBC does not define, you must use the SQL syntax that is specific to each particular data source. This SQL syntax might not be consistent among the data sources that your application uses.

Escape clause syntax

Because ODBC vendor escape clauses are implemented identically across all products and vendors, ODBC defines a short-form escape clause that includes only the extended SQL text. DB2 ODBC supports the following short-form escape clause:

```
{ extended SQL text }
```

extended SQL text
: In ODBC, the string of extended SQL that the ODBC driver translates to data source specific SQL. "ODBC-defined SQL extensions" on page 467 specifies the syntax of the SQL strings that you can use for this parameter.

This short-form escape clause that does not conform to X/Open specifications, but it is widely used among ODBC drivers. In ODBC 3.0, the short ODBC format replaces the deprecated long X/Open format.

DB2 ODBC supports the SQL escape clause X/Open defines with the following long-form syntax:

```
--(*vendor(vendor-identifier),
     product(product-identifier) extended SQL text*)--
```

vendor-identifier
: Vendor identification that is consistent across all of that vendor's SQL products. (For DB2 ODBC, this identifier can be set to either IBM or Microsoft.)

product-identifier
: Identifier for an SQL product. (For DB2 ODBC, this identifier is always set to ODBC.)

extended SQL text
: The same text that the short-form escape clause uses.

Long-form vendor escape clauses are considered deprecated in ODBC 3.0. Although DB2 ODBC supports both long and short formats, you should use the current, short-form escape clauses in your applications.

ODBC-defined SQL extensions

ODBC defines the following SQL extensions (which are not defined by X/Open):

- Extended date, time, and timestamp data
- Outer join
- LIKE predicate
- Call stored procedure
- Extended scalar functions
  - Numeric functions
  - String functions
  - System functions

The following sections describe ODBC-defined syntax that you use for the *extended SQL text* parameter in each ODBC vendor escape clause. DB2 ODBC accepts these SQL extensions as both long-form and short-form vendor escape clauses.

ODBC date, time, and timestamp data

In ODBC, the following extended SQL syntax defines date, time, and timestamp data respectively. You use this syntax in a vendor escape clause to make these definitions portable in your SQL statements.

```
►►──┬─d──┬──'─value─'──────────────────────────────────────────►◄
    ├─t──┤
    └─ts─┘
```

d	Indicates that *value* is a date in the *yyyy-mm-dd* format.
t	Indicates that *value* is a time in the *hh:mm:ss* format.
ts	Indicates that *value* is a timestamp in the *yyyy-mm-dd hh:mm:ss.ffffff* format.
value	Specifies your user data.

Example: You can use either of the following forms of the escape clause to issue a query on the EMPLOYEE table. In this example, a vendor escape clause specifies the data for the predicate in each query.

- Short-form syntax:

```
SELECT * FROM EMPLOYEE WHERE HIREDATE={d '2004-03-29'}
```

- Long-form syntax:

```
SELECT * FROM EMPLOYEE
WHERE HIREDATE=--(*vendor(Microsoft),product(ODBC) d '2004-03-29' *)--
```

You can use the ODBC vendor escape clauses for date, time, and timestamp literals in input parameters with a C data type of SQL_C_CHAR.

To determine if a data source supports date, time, or timestamp data, call SQLGetTypeInfo(). If a data source supports any of these data types, the ODBC driver for that data source supports a corresponding vendor escape clause.

ODBC outer join syntax

In ODBC, the following extended SQL syntax specifies an outer join. Use this syntax in a vendor escape clause to make outer joins portable in your SQL statements.

```
►►──oj──table-name──┬─LEFT──┬──OUTER JOIN──┬─table-name─┬──────────────────────►
                    ├─RIGHT─┤              └─outer-join─┘
                    └─FULL──┘

►──ON──search-condition───────────────────────────────────────────────────────►◄
```

table-name
: Specifies the name of the table that you want to join.

LEFT Performs a left outer join.

RIGHT
: Performs a right outer join.

FULL Performs a full outer join.

table-name
: Specifies the name of the table that you want to join with the previous table.

outer-join
: Specifies the result of an outer join that you want to join with the previous table. (Use the syntax above without the leading keyword oj.)

search-condition
: Specifies the condition on which rows are joined.

Example: You can use either of the following forms of the escape clause to perform an outer join. In this example, a vendor escape clause specifies the outer join in each SQL statement.

- Short-form syntax:

```
SELECT * FROM {oj T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3}
        WHERE T1.C2>20
```

- Long-form syntax:

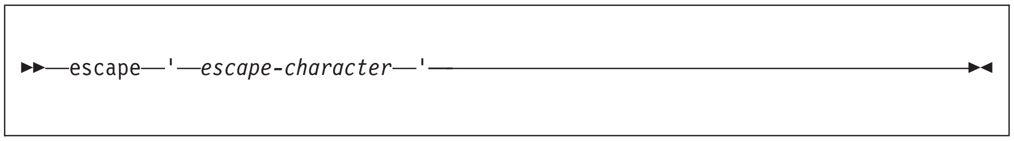
```
SELECT * FROM
     --(*vendor(Microsoft),product(ODBC) oj
        T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3*)--
        WHERE T1.C2>20
```

Important: Not all servers support outer join. To determine if the current server supports outer joins, call SQLGetInfo() twice, first with the *InfoType* argument set to SQL_OUTER_JOINS, and then with the *InfoType* argument set to SQL_OJ_CAPABILITIES.

Like predicate escape clause

In an SQL LIKE predicate, the percent metacharacter (%) matches a string of zero or more characters, and the underscore metacharacter (_) matches any single character. With the predicate escape clause, you can define patterns that contain the actual percent and underscore characters. To specify that you want these characters to represent literal values, you precede them with an escape character.

You define the LIKE predicate escape character with the following syntax in a vendor escape clause:



escape-character
: Specifies any character that is supported by the DB2 rules and that governs the use of the ESCAPE clause.

Example: You can use either of the following forms of the escape clause to include metacharacters as literals in the LIKE predicate. In this example, both statements search for a string that ends with the percent character .

- Short-form syntax:

```
SELECT * FROM EMPLOYEE
WHERE COMMISSION LIKE {escape '!'} '%!%'
```

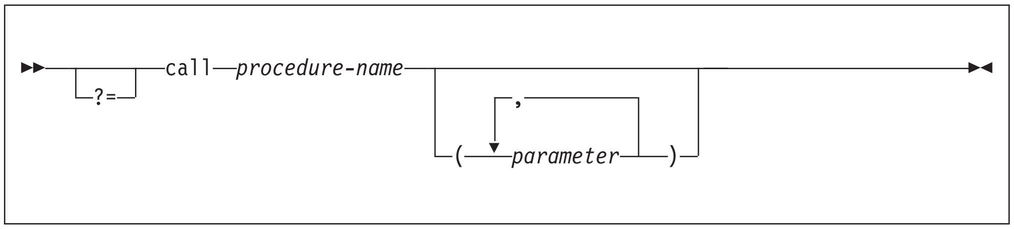
- Long-form syntax:

```
SELECT * FROM EMPLOYEE
WHERE COMMISSION LIKE --(*vendor(Microsoft),product(ODBC) escape '!'*)-- '%!%'
```

To determine if a particular data source supports LIKE predicate escape characters, call SQLGetInfo() with the *InfoType* argument set to SQL_LIKE_ESCAPE_CLAUSE.

Stored procedure CALL

In ODBC, the following extended SQL syntax calls a stored procedure. You use this syntax in a vendor escape clause to make stored procedure calls portable in your SQL statements.



?=
: Specifies that you want DB2 ODBC to return the SQLCODE of the stored procedure call in the first parameter that you specify in SQLBindParameter(). If ?= is not present, you can retrieve the SQLCA with SQLGetSQLCA() .

procedure-name
: Specifies the name of a procedure that is stored at the data source.

parameter
: Specifies a procedure parameter. A procedure can have zero or more parameters.

  Important: Unlike ODBC, DB2 ODBC does not support literals as procedure arguments. You must use parameter markers to specify a procedure parameter.

For more information about stored procedures, see “Using stored procedures” on page 429 or *DB2 Application Programming and SQL Guide*.

Example: You can use either of the following forms of the escape clause to call a stored procedure. In this example, the statements call the procedure NETB94, which uses three parameters.

- Short-form syntax:
  ```
  {CALL NETB94(?,?,?)}
  ```
- Long-form syntax:
  ```
  --(*vendor(Microsoft),product(ODBC) CALL NEBT94(?,?,?)*)--
  ```

To determine if a particular data source supports stored procedure calls, call SQLGetInfo() with the *InfoType* argument set to SQL_PROCEDURES.

ODBC scalar functions

You can use scalar functions such as string length, substring, or trim on columns of result sets, or on columns that restrict rows of a result set. ODBC defines the following extended SQL syntax to call scalar functions. Use this syntax in a vendor escape clause to make portable scalar function calls in your SQL statements.

```
►►──fn──scalar-function──────────────────────────────────────────────►◄
```

scalar-function
: Specifies any function listed in Appendix B, “Extended scalar functions,” on page 493.

Example: You can use either of the following forms of the escape clause to call a scalar function. Both statements in this example use a vendor escape clause in the select list of a query.

- Short-form syntax:
  ```
  SELECT {fn CONCAT(FIRSTNAME,LASTNAME)} FROM EMPLOYEE
  ```
- Long-form syntax:
  ```
  SELECT --(*vendor(Microsoft),product(ODBC) fn CONCAT(FIRSTNAME,LASTNAME) *)--
  FROM EMPLOYEE
  ```

To determine which scalar functions are supported by the current server that is referenced by a specific connection handle, call SQLGetInfo() with the *InfoType* argument set to each of the following values:
- SQL_NUMERIC_FUNCTIONS
- SQL_STRING_FUNCTIONS
- SQL_SYSTEM_FUNCTIONS
- SQL_TIMEDATE_FUNCTIONS

Programming hints and tips

This section provides some hints and tips to help you avoid common problems, improve performance, reduce network flow, and maximize portability when you program a DB2 ODBC application.

Avoiding common problems

The following items present common problems in DB2 ODBC applications:
- The DB2 ODBC initialization file
- Result sets that are too large
- Distinct types

To avoid common problems, adhere to the following guidelines for each DB2 ODBC item.

The DB2 ODBC initialization file

When you alter the DB2 ODBC initialization file, take the following actions:

- Check the coding of square brackets. The square brackets in the initialization file must consist of the correct EBCDIC characters. The open square bracket must use the hexadecimal characters X'AD'. The close square bracket must use the hexadecimal characters X'BD'. DB2 ODBC does not recognize brackets if you code them differently.
- Eliminate sequence numbers. DB2 ODBC does not accept sequence numbers in the initialization file. You must remove all sequence numbers.

|

Result sets that are too large

To limit the number of rows that your application can fetch, set the SQL_ATTR_MAX_ROWS attribute with SQLSetStmtAttr(). You can use this attribute to ensure that a very large result set does not overwhelm your application. This kind of protection is especially important for applications that run on clients with limited memory resources.

Important: The server generates a full result set regardless of the SQL_ATTR_MAX_ROWS attribute value. DB2 ODBC limits only the fetch to SQL_ATTR_MAX_ROWS.

Distinct types

When you use a distinct-type parameter in the predicate of a query statement, you must use a CAST function. With the cast function, cast either the parameter marker to a distinct type, or cast the distinct type to a source type.

Example: Assume that you define the following distinct type and table:

```
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) )
```

Then you issue the following query statement:

```
SELECT  first_name, last_name, phone_num FROM customer
     where cust_num = ?
```

This query fails because the comparison includes incompatible types; the parameter marker cannot be type CNUM.

To successfully execute the statement, issue a query that casts the parameter marker to the distinct type CNUM:

```
SELECT  first_name, last_name, phone_num FROM customer
       where cust_num = cast( ? as cnum )
```

Alternatively, issue a query that casts the data type of the column to the source type INTEGER:

```
SELECT  first_name, last_name, phone_num FROM customer
       where cast( cust_num as integer ) = ?
```

See *DB2 SQL Reference* for more information about parameter markers (PREPARE statement) and casting (CAST function).

Improving application performance

To improve the performance of your DB2 ODBC applications, consider taking the following actions:
- Set isolation levels.
- Disable cursor hold behavior.
- Retrieve result sets efficiently.
- Limit the use of catalog functions.
- Use dynamic statement caching.
- Turn off statement scanning.

Setting isolation levels

You determine the level of locking that is required to execute a statement, and therefore the level of concurrency that is possible, in your application with isolation levels. You need to choose isolation levels for your application that maximize concurrency, and that also ensure data consistency.

Set the minimum isolation level that is possible to maximize concurrency. You can set isolation levels either by statement or by connection:
- SQLSetConnectAttr() with the SQL_ATTR_TXN_ISOLATION attribute specified sets the isolation level at which all statements on a connection handle operate. This isolation level determines the level of concurrency that is possible, and the level of locking that is required to execute any statement on a connection handle.
- SQLSetStmtAttr() with the SQL_ATTR_STMTTXN_ISOLATION attribute sets the isolation level at which an individual statement handle operates. (Although you can set the isolation level on a statement handle, setting the isolation level on the connection handle is recommended.) This isolation level determines the level of concurrency that is possible, and the level of locking that is required to execute the statement.

For more information about setting isolation levels, see "SQLSetConnectAttr() - Set connection attributes" on page 346, "SQLSetStmtAttr() - Set statement attributes" on page 367, and "TXNISOLATION" on page 61.

DB2 ODBC uses resources that are associated with statement handles more efficiently if you set an appropriate isolation level, rather than leaving all statements at the default isolation level. This should be attempted only with a thorough understanding of the locking and isolation levels of the connected DBMS. See *DB2 SQL Reference* and Part 4 of *DB2 Application Programming and SQL Guide* for a complete discussion of isolation levels and their effect.

| ### Disabling cursor hold behavior

DB2 ODBC can more efficiently use resources associated with statement handles if you disable cursor-hold behavior for statements that do not require it.

To disable cursor-hold behavior on a statement handle, call SQLSetStmtAttr() with the SQL_ATTR_CURSOR_HOLD attribute set to SQL_CURSOR_HOLD_OFF. You can also set the cursor-hold behavior for an entire data source through the initialization file. See "CURSORHOLD" on page 54 for more information.

The SQL_ATTR_CURSOR_HOLD statement attribute is the DB2 ODBC equivalent to the CURSOR WITH HOLD clause in SQL. DB2 ODBC cursors exhibit cursor-hold behavior by default.

Important: Many ODBC applications expect a default behavior in which the cursor position is maintained after a commit. Consider such applications before you disable any cursor-hold behavior.

Retrieving data efficiently

Two actions make your application retrieve data sets more efficiently:

- Define the *pcbValue* and *rgbValue* arguments of SQLBindCol() or SQLGetData() contiguously in memory. (This allows DB2 ODBC to fetch both values with one copy operation.)

  To define the *pcbValue* and *rgbValue* arguments contiguously in memory, create a structure that contains both values. For example, the following code creates such a structure:

  ```
  struct {  SQLINTEGER  pcbValue;
            SQLCHAR     rgbValue[MAX_BUFFER];
         } column;
  ```
- Choose an appropriate function with which to retrieve results. Generally the most efficient approach is to bind application variables to result sets with SQLBindCol(). However, in some cases calling SQLGetData() to retrieve results is more efficient. When the data value is large and is variable-length, use SQLGetData() for the following situations:
  – You must retrieve the data in pieces.
  – You might not need to retrieve the data. (That is, retrieval is dependent on another application action.)

Limiting use of catalog functions

In general, try to limit the number of times that you call catalog functions in your application, limit the number of rows that these functions return, and close all open cursors on catalog result sets.

Call each catalog function once and store the information that the function returns in your application to reduce the number of catalog functions that you call.

Specify the following parameters to limit the number of rows that a catalog function returns:

- 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

Close any open cursors (call the SQLCloseCursor() function) for statement handles that are 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.

Recommendation: Plan ahead. Although you might develop and test an application on a data source with hundreds of tables, the final application might sometime need to run on a production database with thousands of tables.

Using dynamic SQL statement caching

To reduce function call overhead, you can prepare a statement once and execute it repeatedly throughout the application.

DB2 servers cache prepared versions of dynamic SQL statements. This dynamic caching allows the DB2 server to reuse previously prepared statements.

To take advantage of dynamic caching, use the same statement handle to execute identical SQL statements. Free this handle only when you no longer need to execute that statement repeatedly.

For example, if your application routinely uses a set of 10 SQL statements, you should allocate 10 statement handles that are associated with each of those statements. Do not free these statement handles until you can no longer execute the statements that are associated with them. You can roll back and commit the transaction without affecting prepared statements. Your application can continue to prepare and execute the statements in a normal manner. The DB2 server determines if a prepare is actually needed.

Turning off statement scanning

To increase performance, allow DB2 ODBC to scan for vendor escape clauses only on handles where escape clauses appear.

By default, DB2 ODBC scans each SQL statement for vendor escape clauses. If your application does not generate SQL statements that contain vendor escape clauses, turn off statement scanning. (For more information about vendor escape clauses see “Using vendor escape clauses” on page 465.)

To turn off statement scanning, set the SQL_ATTR_NOSCAN statement attribute to SQL_NOSCAN_ON. You can set this attribute with either of the following functions:

SQLSetStmtAttr()
When you set the SQL_ATTR_NOSCAN statement attribute to SQL_NOSCAN_ON with SQLSetStmtAttr(), you turn off statement scanning for all SQL statements that are issued on a statement handle.

SQLSetConnectAttr()
When you set the SQL_ATTR_NOSCAN statement attribute to SQL_NOSCAN_ON with SQLSetConnectAttr(), you turn off statement scanning for all SQL statements that are issued on a connection handle.

Reducing network flow

To reduce the network flow that your DB2 ODBC applications generate, consider the following actions:

- Use SQLSetColAttributes() to reduce network flow.
- Disable autocommit.
- Use arrays to send and retrieve data.
- Manipulate large data values at the server.

Using SQLSetColAttributes() to reduce network flow

Each time that you prepare or execute a query statement directly, DB2 ODBC retrieves information about the SQL data type and the size of the data from the data source. If you use SQLSetColAttributes() to provide DB2 ODBC with this information ahead of time, you eliminate the need for DB2 ODBC to query the data source. Elimination of this query can significantly reduce network flow from remote data sources if the result set that comes back contains a very large number (hundreds) of columns.

Requirement: You must provide DB2 ODBC with exact result descriptor information for **all** columns; otherwise, an error occurs when you fetch the data.

SQLSetColAttributes() reduces the network flow best from queries that generate result sets with a large number of columns, but a relatively small number of rows.

Disabling autocommit

Generally, to reduce network flow, you should set the SQL_ATTR_AUTOCOMMIT connection attribute to SQL_AUTOCOMMIT_OFF. Each commit request can generate extra network flow.

Set this attribute to SQL_AUTOCOMMIT_ON only if the application that you are writing needs to treat each statement as a single, complete transaction. See "SQLSetConnectAttr() - Set connection attributes" on page 346 for more information about setting this attribute.

Important: SQL_AUTOCOMMIT_ON is the default setting for this attribute, unless it is otherwise specified in the initialization file. For more information about setting this attribute in the initialization file, see "AUTOCOMMIT" on page 52.

Using arrays to send and retrieve data

Sending multiple data values through the network using arrays rather than individual application variables reduces network flow. For optimum results, use arrays to both send and retrieve data.

"Using arrays to pass parameter values" on page 414 and "Retrieving a result set into an array" on page 417 describe the methods that you can use to send and retrieve data with arrays. Use these methods as much as possible in your application.

Manipulating large data values at the server

Use LOB data types and the functions that support LOB data types for long strings whenever possible. Unlike LONG VARCHAR, LONG VARBINARY, and LONG VARGRAPHIC data types, LOB data values can use LOB locators and functions, such as SQLGetPosition() and SQLGetSubString(), to manipulate large data values at the server.

Maximizing application portability

To maximize the portability of your DB2 ODBC applications, consider the following actions:

- Use column names of function-generated result sets.
- Use SQLDriverConnect() instead of SQLConnect().

Using column position in function-generated result sets

The column names of result sets that are generated by catalog and get-information functions, such as SQLGetInfo(), can change as the X/Open and ISO standards evolve. The position of these columns, however, is fixed.

To maximize the portability of your application, base all dependencies on column position (referred to as the *icol* argument in some functions) rather than on the column name.

Using SQLDriverConnect() instead of SQLConnect()

SQLDriverConnect() overrides any or all of the initialization keyword values that are specified in the DB2 ODBC initialization file for a target data source.

Use SQLDriverConnect() instead of SQLConnect() to make a connection in your application behave independently of the DB2 ODBC initialization file.

Chapter 6. Problem diagnosis

This chapter provides guidelines for working with the DB2 ODBC traces and information about general diagnosis, debugging, and abends. You can obtain traces for DB2 ODBC applications and diagnostics and DB2 ODBC stored procedures.

Tracing

DB2 ODBC provides two traces that differ in purpose:

- An application trace intended for debugging user applications, described in "Application trace."
- A service trace for problem diagnosis, described in "Diagnostic trace" on page 479.

Application trace

The DB2 ODBC application trace is enabled using the APPLTRACE and APPLTRACEFILENAME keywords in the DB2 ODBC initialization file.

The APPLTRACE keyword is intended for customer application debugging. This trace records data information at the DB2 ODBC API interface; it is specifically designed to trace ODBC API calls. The trace is written to the file specified on the APPLTRACEFILENAME keyword.

Recommendation: Use this trace to debug your DB2 ODBC applications.

Specifying the trace file name

You can use the following formats to specify the APPLTRACEFILENAME keyword setting:

- JCL data definition format
- z/OS UNIX environment HFS format

The primary use of the JCL data definition statement format is to write to a z/OS preallocated sequential data set. You can also specify z/OS UNIX HFS files on a DD statement. The z/OS UNIX environment HFS format is used strictly for writing to HFS files.

JCL data definition statement format: The JCL data definition statement format is `APPLTRACEFILENAME="DD:`*`ddname`*`"`. The *ddname* value is the name of the data definition statement that is specified in your job or TSO logon procedure.

Example: Assume the keyword setting is `APPLTRACEFILENAME="DD:APPLDD"`. You can use the following JCL data definition statements in your job or TSO logon procedure to specify the z/OS trace data set.

- Write to preallocated sequential data set USER01.MYTRACE.

  ```
  //APPLDD   DD      DISP=SHR,DSN=USER01.MYTRACE
  ```

- Write to preallocated UNIX HFS file MYTRACE in directory /usr/db2.

  ```
  //APPLDD        DD      PATH='/usr/db2/MYTRACE'
  ```

- Allocate UNIX HFS file MYTRACE in directory /usr/db2 specifying permission for the file owner to read from (SIRUSR) and write to (SIWUSR) the trace file:

  ```
  //APPLDD     DD  PATH='/usr/db2/MYTRACE',
             PATHOPTS=(ORDWR,OCREAT,OTRUNC),
             PATHMODE=(SIRUSR,SIWUSR)
  ```

z/OS UNIX environment HFS format: The z/OS UNIX HFS file name format is `APPLTRACEFILENAME=hfs_filename`. The *hfs_filename* value specifies the path and file name for the HFS file. The HFS file does not have to be preallocated. If the file name does not exist in the specified directory, the file is dynamically allocated.

Example: The following statements use the APPLTRACEFILENAME keyword to specify a z/OS UNIX environment HFS trace file.

- Create and write to HFS file named APPLTRC1 in the fully qualified directory /usr/db2.

  ```
  APPLTRACEFILENAME=/usr/db2/APPLTRC1
  ```
- Create and write to HFS file named APPLTRC1 in the current working directory of the application.

  ```
  APPLTRACEFILENAME=./APPLTRC1
  ```
- Create and write to HFS file named APPLTRC1 in the parent directory of the current working directory.

  ```
  APPLTRACEFILENAME=../APPLTRC1
  ```

Application trace output

Figure 61 on page 479 contains an example of application trace output that shows how DB2 ODBC follows the APIs invoked, indicates values used, data pointers, and so on. Errors are also indicated.

```
SQLAllocHandle( fHandleType=SQL_HANDLE_ENV, hInput=0, phOutput=&6b7e77c )
SQLAllocHandle( phOutput=1 )
    ---> SQL_SUCCESS

SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=1, phOutput=&6b7e778 )
SQLAllocHandle( phOutput=1 )
    ---> SQL_SUCCESS

SQLConnect( hDbc=1, szDSN=Null Pointer, cbDSN=0, szUID=Null Pointer, cbUID=0,
szAuthStr=Null Pointer, cbAuthStr=0 )
SQLConnect( )
    ---> SQL_SUCCESS

SQLAllocHandle( fHandleType=SQL_HANDLE_STMT, hInput=1, phOutput=&6b7e774 )
SQLAllocHandle( phOutput=1 )
    ---> SQL_SUCCESS

SQLExecDirect( hStmt=1, pszSqlStr="SELECT NAME FROM SYSIBM.SYSPLAN", cbSqlStr=-3 )
SQLExecDirect( )
    ---> SQL_SUCCESS

SQLFetch( hStmt=1 )
SQLFetch( )
    ---> SQL_SUCCESS

SQLEndTran( fHandleType=SQL_HANDLE_DBC, fHandle=1, fType=SQL_COMMIT )
SQLEndTran( )
    ---> SQL_SUCCESS

SQLFreeHandle( fHandleType=SQL_HANDLE_STMT, hHandle=1 )
SQLFreeHandle()
    ---> SQL_SUCCESS

SQLDisconnect( hDbc=1 )
SQLDisconnect( )
    ---> SQL_SUCCESS

SQLFreeHandle( fHandleType=SQL_HANDLE_DBC, hHandle=1 )
SQLFreeHandle( )
    ---> SQL_SUCCESS

SQLFreeHandle( fHandleType=SQL_HANDLE_ENV, hHandle=1 )
SQLFreeHandle( )
    ---> SQL_SUCCESS
```

Figure 61. Example application trace output

For more information about how to specify the APPLTRACE and APPLTRACEFILENAME keywords, see “DB2 ODBC initialization file” on page 49.

Diagnostic trace

The DB2 ODBC diagnostic trace captures information to use in DB2 ODBC problem determination. The trace is intended for use under the direction of the IBM Software Support; it is not intended to assist in debugging user written DB2 ODBC applications. You can view this trace to obtain information about the general flow of an application, such as commit information. However, this trace is intended for IBM service information only and is therefore subject to change.

You can activate the diagnostic trace by issuing the DSNAOTRC command or by specifying DIAGTRACE=1 in the DB2 ODBC initialization file.

If you activate the diagnostic trace using the DIAGTRACE keyword in the initialization file, you must also allocate a DSNAOTRC data definition statement in your job or TSO logon procedure. You can use one of the following methods to allocate a DSNAOTRC data definition statement:

- Specify a DSNAOTRC data definition JCL statement in your job or TSO logon procedure.
- Use the TSO/E ALLOCATE command.
- Use dynamic allocation in your ODBC application.

Specifying the diagnostic trace file

The diagnostic trace data can be written to a z/OS sequential data set or a z/OS UNIX environment HFS file.

A z/OS data set must be preallocated with the following data set attributes:
- Sequential data set organization
- Fixed-block 80 record format

When you execute an ODBC application in the z/OS UNIX environment and activate the diagnostic trace using the DIAGTRACE keyword in the initialization file, DB2 writes the diagnostic data to a dynamically allocated file, DD:DSNAOTRC. This file is located in the current working directory of the application if the DSNAOTRC data definition statement is not available to the ODBC application. You can format DD:DSNAOTRC using the DSNAOTRC trace formatting program.

Example: The following JCL examples use a DSNAOTRC data definition JCL statement to specify the diagnostic trace file.

- Write to preallocated sequential data set USER01.DIAGTRC.

  ```
  //DSNAOTRC DD      DISP=SHR,DSN=USER01.DIAGTRC
  ```
- Write to the preallocated z/OS UNIX environment HFS file DIAGTRC in the directory /usr/db2.

  ```
  //DSNAOTRC      DD      PATH='/usr/db2/DIAGTRC'
  ```
- Allocate the z/OS UNIX environment HFS file DIAGTRC in the directory /usr/db2 specifying permission for the file owner to read from (SIRUSR) and write to (SIWUSR) the trace file.

  ```
  //DSNAOTRC        DD   PATH='/usr/db2/DIAGTRC',
                      PATHOPTS=(ORDWR,OCREAT,OTRUNC),
                      PATHMODE=(SIRUSR,SIWUSR)
  ```

For more information about the DIAGTRACE keyword, see "DB2 ODBC initialization file" on page 49.

Using the diagnostic trace command DSNAOTRC

You use the DSNAOTRC command to perform the following tracing tasks:

- Manually start or stop the recording of memory resident diagnostic trace records.
- Query the current status of the diagnostic trace.
- | Capture the memory resident trace table to a z/OS data set or a z/OS UNIX environment HFS file.
- Format the DB2 ODBC diagnostic trace.

Syntax:

```
►►──DSNAOTRC──┬──ON──┬──────────────────┬────────────────────────────┬──────►◄
              │      ├─ -L──buffer size─┤                            │
              │      └─ -I──buffer size─┘                            │
              ├──OFF─────────────────────────────────────────────────┤
              ├──INF─────────────────────────────────────────────────┤
              ├──DMP──trace data set spec────────────────────────────┤
              ├──FMT──┬───────────────────────────────────────────┬──┤
              │       └─input data set spec──┬──────────────────────┬┘  │
              │                              └─output data set spec─┘   │
              └──FLW──input data set spec──┬──────────────────────┬────┘
                                           └─output data set spec─┘
```

Option descriptions:

ON
: Start the DB2 ODBC diagnostic trace.

  -L *buffer size*
  : L = Last. The trace wraps; it captures the last, most current trace records.

    buffer size is the number of bytes to allocate for the trace buffer. This value is required. The buffer size is rounded to a multiple of 65536 (64K).

  -I *buffer size*
  : I = Initial. The trace does not wrap; it captures the initial trace records.

    buffer size is the number of bytes to allocate for the trace buffer. This value is required. The buffer size is rounded to a multiple of 65536 (64K).

OFF
: Stop the DB2 ODBC diagnostic trace.

INF
: Display information about the currently active DB2 ODBC diagnostic trace.

DMP
: Dump the currently active DB2 ODBC diagnostic trace.

  trace data set spec
  : Specifies the z/OS data set or the z/OS UNIX environment HFS file to which DB2 writes the raw DB2 ODBC diagnostic trace data. The data set specification can be either a z/OS data set name, a z/OS UNIX environment HFS file name, or a currently allocated JCL data definition statement name.

FMT
: Generate a formatted detail report of the DB2 ODBC diagnostic trace contents.

FLW
: Generate a formatted flow report of the DB2 ODBC diagnostic trace contents.

  input data set spec
  : The data set that contains the raw DB2 ODBC diagnostic trace data to be formatted. This is the data set that was generated as the result of a DSNAOTRC DMP command or that is generated by the DSNAOTRC data definition statement when the DIAGTRACE initialization keyword enables tracing. The data set specification can be either a z/OS data set name, a z/OS UNIX environment HFS file name, or a currently allocated JCL data

definition statement name. If this parameter is not specified, the DSNAOTRC command attempts to format the memory resident DSNAOTRC that is currently active.

output data set spec
: The data set to which the formatted DB2 ODBC diagnostic trace records are written. The data set specification can be either a z/OS data set name, a z/OS UNIX environment HFS file name, or a currently allocated JCL data definition statement name. If you specify a z/OS data set or a z/OS UNIX environment HFS file that does not exist, DB2 allocates it dynamically. If this parameter is not specified, the output is written to standard output ("`STDOUT`").

Special considerations for the z/OS UNIX environment: You can issue the DSNAOTRC or the DSNAOTRX command from the z/OS UNIX environment command line to activate the diagnostic trace prior to executing an ODBC application. Under the direction of IBM support only, you must store the DSNAOTRC program load modules in a z/OS UNIX environment HFS file.

Use the TSO/E command, OPUTX, to store the DSNAOTRC load modules in an HFS file. The following example uses the OPUTX command to store load module DSNAOTRC from the partitioned data set DB2A.DSNLOAD to the HFS file DSNAOTRC in the directory /usr/db2:

```
OPUTX  'DB2A.DSNLOAD(DSNAOTRC)'  /usr/db2/dsnaotrc
```

After storing the DSNAOTRC program modules in HFS files, follow these steps in the z/OS UNIX environment to activate, dump, and format the diagnostic trace:

1. Enable the shared address space environment variable for the z/OS UNIX shell. Issue the following export statement at the command line or specify it in your $HOME/.profile file:

   ```
   export _BPX_SHAREAS=YES
   ```

   Setting this environment variable allows the OMVS command and the z/OS UNIX shell to run in the same TSO address space.
2. Go to the directory that contains the DSNAOTRC load modules.
3. Verify that execute permission is established for the DSNAOTRC load modules. If execute permission was not granted, use the chmod command to set execute permission for each of the load modules.
4. Issue `dsnaotrc on`. The options for activating the diagnostic trace are optional.
5. Execute the ODBC application.
6. Issue `dsnaotrc dmp "`*`raw_trace_file`*`"`. The *raw_trace_file* value is the name of the output file to which DB2 writes the raw diagnostic trace data.
7. Issue `dsnaotrc off` to deactivate the diagnostic trace.
8. Issue `dsnaotrc  fmt  "`*`raw_trace_file`*`"  "`*`fmt_trace_file`*`"` to format the raw trace data records from input file `"`*`raw_trace_file`*`"` to output file `"`*`fmt_trace_file`*`"`.

After successfully formatting the diagnostic trace data, delete the DSNAOTRC program modules from your z/OS UNIX environment directory. Do not attempt to maintain a private copy of the DSNAOTRC program modules in your HFS directory.

Example: Each of the following statements show how to code the trace data set specification.

- Currently allocated JCL data definition statement name TRACEDD

```
DSNAOTRC DMP DD:TRACEDD
```

- Sequential data set USER01.DIAGTRC

```
DSNAOTRC DMP "USER01.DIAGTRC"
```

- z/OS UNIX environment HFS file that is named DIAGTRC in directory /usr/db2

```
DSNAOTRC DMP "/usr/db2/DIAGTRC"
```

Example: Each of the following statements show how to code the input data set specification.

- Currently allocated JCL data definition statement name INPDD.

```
DSNAOTRC FLW DD:INPDD output-dataset-spec
```

- Sequential data set USER01.DIAGTRC.

```
DSNAOTRC FLW "USER01.DIAGTRC" output-dataset-spec
```

- z/OS UNIX environment HFS file DIAGTRC in directory /usr/db2.

```
DSNAOTRC FLW "/usr/db2/DIAGTRC" output-dataset-spec
```

Example: Each of the following statements show how to code the output data set specification.

- Currently allocated JCL data definition statement name OUTPDD.

```
DSNAOTRC FLW input-dataset-spec DD:OUTPDD
```

- Sequential data set USER01.TRCFLOW.

```
DSNAOTRC FLW input-dataset-spec "USER01.TRCFLOW"
```

- z/OS UNIX environment HFS file TRCFLOW in directory /usr/db2.

```
DSNAOTRC FLW input-dataset-spec "/usr/db2/TRCFLOW"
```

Stored procedure trace

This section describes the steps required to obtain an application trace or a diagnostic trace of a DB2 ODBC stored procedure. DB2 ODBC stored procedures run in either a DB2-established stored procedures address space or a WLM-established address space. Both the main application that calls the stored procedure (client application), and the stored procedure itself, can be either a DB2 ODBC application or a standard DB2 precompiled application.

If the client application and the stored procedure are DB2 ODBC application programs, you can trace:
- A client application only
- A stored procedure only
- Both the client application and stored procedure

More than one address spaces can not share write access to a single data set. Therefore, you must use the appropriate JCL DD statements to allocate a unique trace data set for each stored procedures address space that uses the DB2 ODBC application trace or diagnostic trace.

Tracing a client application

This section explains how to obtain an application trace and a diagnostic trace for a client application.

Application trace: Follow these steps to obtain an application trace.

1. Set APPLTRACE=1 and APPLTRACEFILENAME=″DD:*DDNAME*″ in the common section of the DB2 ODBC initialization file as follows:

```
[COMMON]
APPLTRACE=1
APPLTRACEFILENAME="DD:APPLTRC"
```

DDNAME is the name of a data definition statement specified in the JCL for the application job or your TSO logon procedure.

2. Specify an APPLTRC data definition statement in the JCL for the application job or your TSO logon procedure. The data definition statement references a preallocated z/OS sequential data set with DCB attributes `RECFM=VBA,LRECL=137`, a z/OS UNIX environment HFS file to contain the client application trace, as shown in the following examples:

```
//APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC
```

```
//APPLTRC DD PATH='/u/cli/appltrc'
```

Diagnostic trace: When tracing only the client application, you can activate the diagnostic trace by using the DIAGTRACE keyword in the DB2 ODBC initialization file, the DSNAOTRC command. See "Diagnostic trace" on page 479 for information about obtaining a diagnostic trace of the client application.

Tracing a stored procedure

This section explains how to obtain an application trace and a diagnostic trace for a stored procedure.

Application trace: Follow these steps to obtain an application trace.

1. Set APPLTRACE=1 and `APPLTRACEFILENAME="DD:`*`DDNAME`*`"` in the common section of the DB2 ODBC initialization file as follows:

```
[COMMON]
APPLTRACE=1
APPLTRACEFILENAME="DD:APPLTRC"
```

DDNAME is the name of a data definition statement that is specified in the JCL for the stored procedures address space.

2. Specify an JCL DD statement in the JCL for the stored procedures address space The data definition statement references a preallocated sequential data set with DCB attributes `RECFM=VBA,LRECL=137` or a z/OS UNIX environment HFS file to contain the client application trace, as shown in the following examples:

```
//APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC
```

```
//APPLTRC DD PATH='/u/cli/appltrc'
```

Diagnostic trace: Follow these steps to obtain a diagnostic trace.

1. Set DIAGTRACE=1, DIAGTRACE_BUFFER_SIZE=*nnnnnnn*, and DIAGTRACE_NO_WRAP=0 or 1 in the common section of the DB2 ODBC initialization file. For example:

```
[COMMON]
DIAGTRACE=1
DIAGTRACE_BUFFER_SIZE=2000000
DIAGTRACE_NO_WRAP=1
```

nnnnnnn is the number of bytes to allocate for the diagnostic trace buffer.

2. Specify a z/OS DSNAOINI data definition statement in the JCL for the stored procedures address space. The data definition statement references the DB2 ODBC initialization file, as shown in the following examples:

```
//DSNAOINI DD DISP=SHR,DSN=CLI.DSNAOINI
```

```
//DSNAOINI DD PATH='/u/cli/dsnaoini'
```

3. Specify a DSNAOTRC data definition statement in the JCL for the stored procedures space. The data definition statement references a preallocated sequential data set with DCB attributes `RECFM=FB,LRECL=80`, or a z/OS UNIX environment HFS file to contain the unformatted diagnostic data, as shown in the following examples:

   ```
   //DSNAOTRC DD DISP=SHR,DSN=CLI.DIAGTRC
   ```

   ```
   //DSNAOTRC DD PATH='/u/cli/diagtrc'
   ```

4. Execute the client application that calls the stored procedure.
5. After the DB2 ODBC stored procedure executes, stop the stored procedures address space.
   - For DB2-established address spaces, use the DB2 command, STOP PROCEDURE.
   - For WLM-established address spaces operating in WLM goal mode, use the z/OS command, "`VARY WLM,APPLENV=`*`name`*`,QUIESCE`". *name* is the WLM application environment name.
   - For WLM-established address spaces operating in WLM compatibility mode, use the z/OS command, "`CANCEL` *`address-space-name`*" where *address-space-name* is the name of the WLM-established address space.
6. You can submit either the formatted or unformatted diagnostic trace data to the IBM Software Support. To format the raw trace data at your site, run the DSNAOTRC FMT or DSNAOTRC FLW commands against the diagnostic trace data set.

Tracing both a client application and a stored procedure

This section explains how to obtain an application trace and a diagnostic trace for both a client application and a stored procedure.

Application trace: Follow these steps to obtain an application trace.

1. Set APPLTRACE=1 and APPLTRACEFILENAME="DD:*DDNAME*" in the common section of the DB2 ODBC initialization file as follows:

   ```
   [COMMON]
   APPLTRACE=1
   APPLTRACEFILENAME="DD:APPLTRC"
   ```

   DDNAME is the name of the data definition statement specified in both the JCL for the client application job and the stored procedures address space.
2. Specify a APPLTRC data definition statement in the JCL for the client application. The data definition statement references a preallocated sequential data set with DCB attributes `RECFM=VBA,LRECL=137`, or a z/OS UNIX environment HFS file to contain the client application trace, as shown in the following examples:

   ```
   //APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC.CLIENT
   ```

   ```
   //APPLTRC DD PATH='/u/cli/appltrc.client'
   ```

   You must allocate a separate application trace data set, or an HFS file for the client application. Do not attempt to write to the same application trace data set or HFS file used for the stored procedure.
3. Specify a APPLTRC data definition statement in the JCL for the stored procedures address space. The data definition statement references a preallocated sequential data set, or a z/OS UNIX environment HFS file to contain the stored procedure application trace, as shown in the following examples:

   ```
   //APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC.SPROC
   ```

```
//APPLTRC DD PATH='/u/cli/appltrc.sproc'
```

You must allocate a separate trace data set or HFS file for the stored procedure. Do not attempt to write to the same application trace data set or HFS file used for the client application.

Diagnostic trace: Follow these steps to obtain a diagnostic trace.

1. Set DIAGTRACE=1, DIAGTRACE_BUFFER_SIZE=*nnnnnnn*, and DIAGTRACE_NO_WRAP=0 or 1 in the common section of the DB2 ODBC initialization file. For example:
   ```
   [COMMON]
   DIAGTRACE=1
   DIAGTRACE_BUFFER_SIZE=2000000
   DIAGTRACE_NO_WRAP=1
   ```
   nnnnnnn is the number of bytes to allocate for the diagnostic trace buffer.
2. Specify a z/OS DSNAOINI data definition statement in the JCL for the stored procedures address space. The data definition statement references the DB2 ODBC initialization file, as shown in the following examples:
   ```
   //DSNAOINI DD DISP=SHR,DSN=CLI.DSNAOINI
   ```
   ```
   //DSNAOINI DD PATH='/u/cli/dsnaoini'
   ```
3. Specify a DSNAOTRC data definition statement in JCL for the client application job. The data definition statement references a preallocated sequential data set with DCB attributes `RECFM=FB,LRECL=80`, or a z/OS UNIX environment HFS file to contain the unformatted diagnostic data, as shown in the following examples:
   ```
   //DSNAOTRC DD DISP=SHR,DSN=CLI.DIAGTRC.CLIENT
   ```
   ```
   //DSNAOTRC DD PATH='/u/cli/diagtrc.client'
   ```
4. Specify a DSNAOTRC data definition statement in the JCL for the stored procedures address space. The data definition statement references a preallocated sequential data set with DCB attributes `RECFM=FB,LRECL=80`, or a z/OS UNIX environment HFS file to contain the stored procedure's unformatted diagnostic data, as shown in the following examples:
   ```
   //DSNAOTRC DD DISP=SHR,DSN=CLI.DIAGTRC.SPROC
   ```
   ```
   //DSNAOTRC DD PATH='/u/cli/diagtrc.sproc'
   ```
5. Execute the client application that calls the stored procedure.
6. After the DB2 ODBC stored procedure executes, stop the stored procedures address space.
   - For DB2-established address spaces, use the DB2 command, STOP PROCEDURE.
   - For WLM-established address spaces operating in WLM goal mode, use the z/OS command, `"VARY WLM,APPLENV=`*`name`*`,QUIESCE"`. *name* is the WLM application environment name.
   - For WLM-established address spaces operating in WLM compatibility mode, use the z/OS command, `"CANCEL `*`address-space-name`*`"`. Where *address-space-name* is the name of the WLM-established address space.
7. You can submit either the formatted or unformatted diagnostic trace data to the IBM Software Support. To format the raw trace data at your site, run the DSNAOTRC FMT or DSNAOTRC FLW command against the client application's diagnostic trace data set and the stored procedure's diagnostic trace data set.

Debugging DB2 ODBC applications

You can debug DB2 UDB for z/OS ODBC applications debug tool shipped with your the C or C++ language compiler. For detailed instructions on debugging DB2 stored procedures, including DB2 ODBC stored procedures, see Part 6 of *DB2 Application Programming and SQL Guide*.

Abnormal termination

Language Environment reports DB2 ODBC abends because DB2 ODBC runs under Language Environment. Typically, Language Environment reports the type of abend that occurs and the function that is active in the address space at the time of the abend.

DB2 ODBC has no facility for abend recovery. When an abend occurs, DB2 ODBC terminates. DBMSs follow the normal recovery process for any outstanding DB2 unit of work.

″CEE″ is the prefix for all Language Environment messages. If the prefix of the active function is ″CLI″, then DB2 ODBC had control during the abend which indicates that this can be a DB2 ODBC, a DB2, or a user error.

The following example shows an abend:

```
CEE3250C The system or user abend S04E  R=00000000 was issued.
         From entry point CLI_mvsCallProcedure(CLI_CONNECTINFO*,...
         +091A2376 at address 091A2376...
```

In this message, you can determine what caused the abend as follows:
- ″CEE″ indicates that Language Environment is reporting the abend.
- The entry point shows that DB2 ODBC is the active module.
- Abend code ″S04E″ means that this is a DB2 system abend.

For more information about debugging, see *z/OS Language Environment Debugging Guide*. For more information about the DB2 recovery process, see Part 4 (Volume 1) of *DB2 Administration Guide*.

Internal error code

DB2 ODBC provides an internal error code for ODBC diagnosis that is intended for use under the guidance of IBM Software Support. This unique error location, ERRLOC, is a good tool for APAR searches. The following example of a failed SQLAllocHandle() (with *HandleType* set to SQL_HANDLE_DBC) shows an error location:

```
DB2 ODBC Sample SQLError Information
DB2 ODBC Sample SQLSTATE          : 58004
DB2 ODBC Sample Native Error Code : -99999
DB2 ODBC Sample Error message text:
  {DB2 for z/OS}{ODBC Driver}  SQLSTATE=58004  ERRLOC=2:170:4;
  RRS "IDENTIFY" failed using DB2 system:V81A,
  RC=08 and REASON=00F30091
```

Appendix A. DB2 ODBC and ODBC

This appendix explains the differences between DB2 ODBC and ODBC in the following areas:
- "DB2 ODBC and ODBC drivers"
- "ODBC APIs and data types" on page 490
- "Isolation levels" on page 492

For a complete list of functions that DB2 ODBC and ODBC support, see Table 12 on page 65.

DB2 ODBC and ODBC drivers

This section discusses the support that is provided by the ODBC driver, and how it differs from DB2 ODBC.

Figure 62 compares DB2 ODBC and the DB2 ODBC driver. The left side of this figure depicts an ODBC driver under the ODBC driver manager. The right side of this figure depicts DB2 ODBC, a callable interface that is designed for DB2-specific applications.

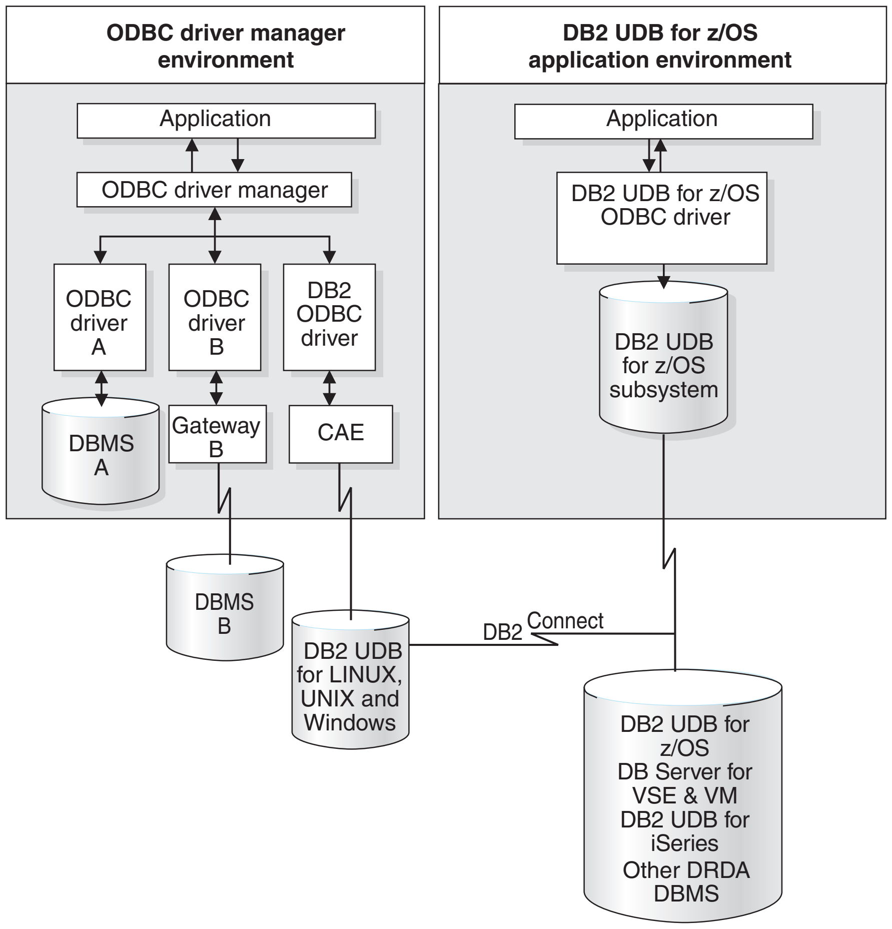


Figure 62. DB2 ODBC and ODBC

In an ODBC environment, the driver manager provides the interface to the application. It also dynamically loads the necessary driver for the database server to which the application connects. It is the driver that implements the ODBC function set, with the exception of some extended functions that are implemented by the driver manager.

The DB2 ODBC driver does not execute in this environment. Rather, DB2 ODBC is a self-sufficient driver which supports a subset of the functions that the ODBC driver provides.

DB2 ODBC applications interact directly with the ODBC driver which executes within the application address space. Applications do not interface with a driver manager. The capabilities that are provided to the application are a subset of the Microsoft ODBC Version 2 specifications.

ODBC APIs and data types

Table 231 summarizes the ODBC Version 3 application programming interfaces, ODBC SQL data types and ODBC C data types and whether those functions and data types are supported by DB2 ODBC. Table 12 on page 65 provides a complete list of functions supported by DB2 ODBC and ODBC 3.0.

Table 231. DB2 ODBC support

ODBC features	DB2 ODBC
Core level functions	All, except for: • SQLDrivers() • SQLGetDescField() • SQLSetDescField() • SQLGetDescRec() • SQLSetDescRec() • SQLCopyDesc() • SQLFetchScroll()
Level 1 functions	All, except for: • SQLSetPos() • SQLBulkOperations() • SQLBrowseConnect()
Level 2 functions	All
Additional DB2 ODBC functions	• SQLSetConnection() • SQLGetEnvAttr() • SQLSetColAttributes() • SQLGetLength() • SQLGetPosition() • SQLGetSubString() • SQLGetSQLCA()
Minimum SQL data types	• SQL_CHAR • SQL_LONGVARCHAR • SQL_VARCHAR
Core SQL data types	• SQL_DECIMAL • SQL_NUMERIC • SQL_SMALLINT • SQL_INTEGER • SQL_REAL • SQL_FLOAT • SQL_DOUBLE

Table 231. DB2 ODBC support (continued)

ODBC features	DB2 ODBC
Extended SQL data types	• SQL_BIT • SQL_TINYINT • SQL_BIGINT (NOT SUPPORTED) • SQL_BINARY • SQL_BLOB • SQL_BLOB_LOCATOR • SQL_CLOB • SQL_CLOB_LOCATOR • SQL_DBCLOB • SQL_DBCLOB_LOCATOR • SQL_LONGVARBINARY • SQL_ROWID • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY
ODBC Version 3 SQL data types	• SQL_GRAPHIC • SQL_LONGVARGRAPHIC • SQL_VARGRAPHIC
Core C data types	• SQL_C_CHAR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG (SLONG, ULONG) • SQL_C_SHORT (SSHORT, USHORT)
Extended C data types	• SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT
ODBC Version 3 C data types	• SQL_C_DBCHAR
Return codes	• SQL_SUCCESS • SQL_SUCCESS_WITH_INFO • SQL_NEED_DATA • SQL_NO_DATA_FOUND • SQL_ERROR • SQL_INVALID_HANDLE
SQLSTATEs	Mapped to X/Open SQLSTATEs with additional IBM SQLSTATEs
Multiple connections per application	Supported but type 1 connections, SQL_ATTR_CONNECTTYPE = SQL_CONCURRENT_TRANS. Must be on a transaction boundary prior to SQLConnect() or SQLSetConnection().

For more information about ODBC, see *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*.

Isolation levels

The following table maps IBM RDBMS isolation levels to ODBC transaction isolation levels. The SQLGetInfo() function indicates which isolation levels are available.

Table 232. 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)
Restriction: SQLSetConnectAttr() and SQLSetStmtAttr() return SQL_ERROR with an SQLSTATE of **HY**009 if you try to set an unsupported isolation level.	

Appendix B. Extended scalar functions

The following functions are defined by ODBC using vendor escape clauses. Each function can be called using the escape clause syntax, or calling the equivalent DB2 function.

These functions are presented in the following categories:
- “String functions”
- “Date and time functions” on page 494
- “System functions” on page 494

For more information about vendor escape clauses, see “ODBC scalar functions” on page 470.

All errors that are detected by the following functions return SQLSTATE **38**552 when you are connected to a DB2 UDB for Linux, UNIX and Windows server. 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 ODBC and defined by ODBC using vendor escape clauses. The following rules apply to input strings for these functions:

- Character string literals used as arguments to scalar functions must be enclosed in 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, or SQL_LONGVARCHAR.
- 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.

ASCII(*string_exp* **)**
Returns the ASCII code value of the leftmost character of *string_exp* as an integer.

CONCAT(*string_exp1***,** *string_exp2* **)**
Returns a character string that is the result of concatenating *string_exp2* to *string_exp1*.

INSERT(*string_exp1, start, length, string_exp2* **)**
Returns a character string where *length* number of characters beginning at *start* is replaced by *string_exp2* which contains *length* characters.

LEFT(*string_exp, count* **)**
Returns the leftmost *count* of characters of *string_exp*.

LENGTH(*string_exp* **)**
Returns the number of characters in *string_exp*, excluding trailing blanks and the string termination character.

REPEAT(*string_exp***,** *count* **)**
Returns a character string composed of *string_exp* repeated *count* times.

RIGHT(*string_exp***,** *count* **)**
Returns the rightmost count of characters of *string_exp*.

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.

Date and time functions

The date and time functions in this section are supported by DB2 ODBC and defined by ODBC using vendor escape clauses. The following rules apply to these functions:

- 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.

CURDATE()
Returns the current date as a date value.

CURTIME()
Returns the current local time as a time value.

DAYOFMONTH(*date_exp* **)**
Returns the day of the month in *date_exp* as an integer value in the range of 1-31.

HOUR(*time_exp* **)**
Returns the hour in *time_exp* as an integer value in the range of 0-23.

MINUTE(*time_exp* **)**
Returns the minute in *time_exp* as integer value in the range of 0-59.

MONTH(*date_exp* **)**
Returns the month in *date_exp* as an integer value in the range of 1-12.

NOW()
Returns the current date and time as a timestamp value.

SECOND(*time_exp* **)**
Returns the second in *time_exp* as an integer value in the range of 0-59.

System functions

The system functions in this section are supported by DB2 ODBC 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.

DATABASE()
: Returns the name of the database corresponding to the connection handle (*hdbc*). (The name of the database is also available using SQLGetInfo() by specifying the information type SQL_DATABASE_NAME.)

IFNULL(*exp*, *value* **)**
: If *exp* is null, *value* is returned. If *exp* is not null, *exp* is returned. The possible data types of *value* must be compatible with the data type of *exp*.

USER()
: Returns the user's authorization name. (The user's authorization name is also available using SQLGetInfo() by specifying the information type SQL_USER_NAME.)

Appendix C. SQLSTATE cross reference

Table 233 is a cross-reference of all the SQLSTATEs listed in the 'Diagnostics' section of each function description in Chapter 4, “Functions,” on page 63.

This table does not include SQLSTATEs that were remapped between ODBC 2.0 and ODBC 3.0, although deprecated functions continue to return these values. For a list of SQLSTATEs that were changed in ODBC 3.0 see “SQLSTATE mappings” on page 528. For a list of deprecated functions see “Mapping deprecated functions” on page 525.

Important: DB2 ODBC can also return SQLSTATEs generated by the server that are not listed in this table. If the returned SQLSTATE is not listed here, see the documentation for the server for additional SQLSTATE information.

Table 233. SQLSTATE cross reference

SQLSTATE	Description	Functions
01000	Warning.	• SQLAllocHandle() • SQLCloseCursor() • SQLColAttribute() • SQLDescribeParam() • SQLEndTran() • SQLFreeHandle() • SQLGetConnectAttr() • SQLGetStmtAttr() • SQLSetConnectAttr() • SQLSetStmtAttr()
01002	Disconnect error.	• SQLDisconnect()
01004	Data truncated.	• SQLColAttribute() • SQLDataSources() • SQLDescribeCol() • SQLDriverConnect() • SQLExtendedFetch() • SQLFetch() • SQLGetConnectAttr() • SQLGetCursorName() • SQLGetData() • SQLGetDiagRec() • SQLGetInfo() • SQLGetStmtAttr() • SQLGetSubString() • SQLNativeSql() • SQLPutData() • SQLSetColAttributes()
01504	The UPDATE or DELETE statement does not include a WHERE clause.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
01S00	Invalid connection string attribute.	• SQLDriverConnect()
01S01	Error in row.	• SQLExtendedFetch()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
01S02	Option value changed.	• SQLDriverConnect() • SQLSetConnectAttr() • SQLSetStmtAttr()
07001	Wrong number of parameters.	• SQLExecDirect() • SQLExecute()
07002	Too many columns.	• SQLExtendedFetch() • SQLFetch()
07005	The statement did not return a result set.	• SQLColAttribute() • SQLDescribeCol()
07006	Invalid conversion.	• SQLBindParameter() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLGetLength() • SQLGetPosition() • SQLGetSubString()
08001	Unable to connect to data source.	• SQLConnect()
08002	Connection in use.	• SQLConnect()
08003	Connection is closed.	• SQLAllocHandle() • SQLDisconnect() • SQLEndTran() • SQLFreeHandle() • SQLGetConnectAttr() • SQLGetInfo() • SQLNativeSql() • SQLSetConnectAttr() • SQLSetConnection()
08004	The application server rejected establishment of the connection.	• SQLConnect()
08007	Connection failure during transaction.	• SQLEndTran()
08S01	Communication link failure.	• SQLSetConnectAttr() • SQLSetStmtAttr()
0F001	The LOB token variable does not currently represent any value.	• SQLGetLength() • SQLGetPosition() • SQLGetSubString()
21S01	Insert value list does not match column list.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
21S02	Degrees of derived table does not match column list.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
22001	String data right truncation.	• SQLExecDirect() • SQLExecute() • SQLPutData()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
22002	Invalid output or indicator buffer specified.	• SQLExtendedFetch() • SQLFetch() • SQLGetData()
22008	Invalid datetime format or datetime field overflow.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLPutData()
22011	A substring error occurred.	• SQLGetSubString()
22012	Division by zero is invalid.	• SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch()
22018	Error in assignment.	• SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLPutData()
23000	Integrity constraint violation.	• SQLExecDirect() • SQLExecute()
24000	Invalid cursor state.	• SQLCloseCursor() • SQLColumnPrivileges() • SQLColumns() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLGetData() • SQLGetStmtAttr() • SQLGetTypeInfo() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedureColumns() • SQLProcedures() • SQLSetColAttributes() • SQLSetStmtAttr() • SQLSpecialColumns() • SQLStatistics() • SQLTablePrivileges() • SQLTables()
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.	• SQLExecDirect() • SQLExecute()
25000 **25**501	Invalid transaction state.	• SQLDisconnect()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
34000	Invalid cursor name.	• SQLExecDirect() • SQLExecute() • SQLPrepare() • SQLSetCursorName()
37*xxx*	Invalid SQL syntax.	• SQLExecDirect() • SQLExecute() • SQLNativeSql() • SQLPrepare()
40001	Transaction rollback.	• SQLEndTran() • SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
40003	Communication link failure.	• SQLBindCol() • SQLBindParameter() • SQLCancel() • SQLColumnPrivileges() • SQLColumns() • SQLDescribeCol() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLFreeStmt() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetInfo() • SQLGetLength() • SQLGetPosition() • SQLGetSubString() • SQLGetTypeInfo() • SQLMoreResults() • SQLNumParams() • SQLNumResultCols() • SQLParamData() • SQLParamOptions() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedureColumns() • SQLProcedures() • SQLPutData() • SQLRowCount() • SQLSetColAttributes() • SQLSetCursorName() • SQLSpecialColumns() • SQLStatistics() • SQLTablePrivileges() • SQLTables()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
42*xxx*	Syntax error or access rule violation	• SQLExecDirect() • SQLExecute() • SQLPrepare()
42000	Invalid SQL syntax.	• SQLNativeSql()
425*xx*	Syntax error or access rule violation	• SQLExecDirect() • SQLExecute() • SQLPrepare()
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() • SQLExecute()
42S01[1]	Database object already exists.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
42S02	Database object does not exist.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
42S11	Index already exists.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
42S12	Index not found.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
42S21	Column already exists.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
42S22	Column not found.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
44000	Integrity constraint violation.	• SQLExecDirect() • SQLExecute()
54028	The maximum number of concurrent LOB handles has been reached.	• SQLFetch()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
58004	Unexpected system failure.	• SQLBindCol() • SQLBindParameter() • SQLConnect() • SQLDriverConnect() • SQLDataSources() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLFreeStmt() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetInfo() • SQLGetLength() • SQLGetPosition() • SQLGetSubString() • SQLMoreResults() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetCursorName()
HY000[2]	General error.	• SQLAllocHandle() • SQLCloseCursor() • SQLColAttribute() • SQLDescribeParam() • SQLEndTran() • SQLFreeHandle() • SQLGetConnectAttr() • SQLGetStmtAttr() • SQLSetColAttributes() • SQLSetConnection() • SQLSetStmtAttr()
HY001	Memory allocation failure.	All functions.
HY002	Invalid column number.	• SQLBindCol() • SQLColAttribute() • SQLDescribeCol() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLSetColAttributes()
HY003	Program type out of range.	• SQLBindCol() • SQLBindParameter() • SQLGetData() • SQLGetLength() • SQLGetSubString()
HY004	Invalid SQL data type.	• SQLBindParameter() • SQLGetTypeInfo()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
HY009	Invalid use of a null pointer.	• SQLAllocHandle() • SQLBindParameter() • SQLColumnPrivileges() • SQLExecDirect() • SQLForeignKeys() • SQLGetData() • SQLGetFunctions() • SQLGetInfo() • SQLGetLength() • SQLGetPosition() • SQLNativeSql() • SQLNumParams() • SQLNumResultCols() • SQLPrepare() • SQLPutData() • SQLSetCursorName() • SQLSetConnectAttr() • SQLSetEnvAttr() • SQLSetStmtAttr()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
HY010	Function sequence error.	• SQLBindCol() • SQLBindParameter() • SQLCloseCursor() • SQLColAttribute() • SQLColumns() • SQLDescribeCol() • SQLDescribeParam() • SQLDisconnect() • SQLEndTran() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLFreeHandle() • SQLFreeStmt() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetStmtAttr() • SQLGetTypeInfo() • SQLMoreResults() • SQLNumParams() • SQLNumResultCols() • SQLParamData() • SQLParamOptions() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedureColumns() • SQLProcedures() • SQLPutData() • SQLRowCount() • SQLSetColAttributes() • SQLSetConnectAttr() • SQLSetCursorName() • SQLSetStmtAttr() • SQLSpecialColumns() • SQLStatistics() • SQLTablePrivileges() • SQLTables()
HY011	Operation invalid at this time.	• SQLSetConnectAttr() • SQLSetEnvAttr() • SQLSetStmtAttr()
HY012	Invalid transaction code.	• SQLEndTran()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
HY013	Unexpected memory handling error.	• SQLAllocHandle() • SQLBindCol() • SQLBindParameter() • SQLCancel() • SQLCloseCursor() • SQLConnect() • SQLDataSources() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLFreeHandle() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetLength() • SQLGetPosition() • SQLGetStmtAttr() • SQLGetSubString() • SQLMoreResults() • SQLNumParams() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetColAttributes() • SQLSetCursorName()
HY014	No more handles.	• SQLAllocHandle() • SQLColumnPrivileges() • SQLColumns() • SQLExecDirect() • SQLExecute() • SQLForeignKeys() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedureColumns() • SQLProcedures() • SQLSpecialColumns() • SQLStatistics() • SQLTablePrivileges() • SQLTables()
HY015	No cursor name available.	• SQLGetCursorName()
HY019	Numeric value out of range.	• SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLPutData()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
HY024	Invalid argument value.	• SQLConnect() • SQLGetSubString() • SQLSetConnectAttr() • SQLSetEnvAttr() • SQLSetStmtAttr()
HY090	Invalid string or buffer length.	• SQLBindCol() • SQLBindParameter() • SQLColAttribute() • SQLColumnPrivileges() • SQLColumns() • SQLConnect() • SQLDataSources() • SQLDescribeCol() • SQLDriverConnect() • SQLExecDirect() • SQLParamData() • SQLForeignKeys() • SQLGetConnectAttr() • SQLGetCursorName() • SQLGetData() • SQLGetInfo() • SQLGetPosition() • SQLGetStmtAttr() • SQLGetSubString() • SQLNativeSql() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedures() • SQLProcedureColumns() • SQLPutData() • SQLSetColAttributes() • SQLSetConnectAttr() • SQLSetCursorName() • SQLSetEnvAttr() • SQLSetStmtAttr() • SQLSpecialColumns() • SQLStatistics() • SQLTables() • SQLTablePrivileges()
HY091	Descriptor type out of range.	• SQLColAttribute()
HY092	Option type out of range.	• SQLAllocHandle() • SQLEndTran() • SQLFreeStmt() • SQLGetConnectAttr() • SQLGetCursorName() • SQLGetEnvAttr() • SQLGetStmtAttr() • SQLSetConnectAttr() • SQLSetEnvAttr() • SQLSetStmtAttr()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
HY093	Invalid parameter number.	• SQLBindParameter() • SQLDescribeParam()
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 or scale value.	• SQLBindParameter() • SQLSetColAttributes()
HY105	Invalid parameter type.	• SQLBindParameter()
HY106	Fetch type out of range.	• SQLExtendedFetch()
HY107	Row value out of range.	• SQLParamOptions()
HY109	Invalid cursor position.	• SQLGetStmtAttr()
HY110	Invalid driver completion.	• SQLDriverConnect()
HY501	Invalid data source name.	• SQLConnect()
HY506	Error closing a file.	• SQLFreeHandle()
HYC00	Driver not capable.	• SQLBindCol() • SQLBindParameter() • SQLColAttribute() • SQLColumnPrivileges() • SQLColumns() • SQLDescribeCol() • SQLDescribeParam() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLGetConnectAttr() • SQLGetData() • SQLGetInfo() • SQLGetLength() • SQLGetPosition() • SQLGetStmtAttr() • SQLGetSubString() • SQLPrimaryKeys() • SQLProcedureColumns() • SQLProcedures() • SQLSetConnectAttr() • SQLSetEnvAttr() • SQLSetStmtAttr() • SQLSpecialColumns() • SQLStatistics() • SQLTables() • SQLTablePrivileges()

Table 233. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions

Notes:

1. **42**S*xx* SQLSTATEs replace **S0**0*xx* SQLSTATEs.
2. **HY***xxx* SQLSTATEs replace **S1***xxx* SQLSTATEs.

Appendix D. Data conversion

This appendix 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, see Table 4 on page 25 and Table 5 on page 27. Supported conversions are shown in Table 8 on page 29 and Table 9 on page 32.

Identifiers for date, time, and timestamp data types changed in ODBC 3.0. See “Changes to datetime data types” on page 529 for the data type mappings.

SQL data type attributes

This section presents the following attributes for each SQL data type:

- “Precision of SQL data types”
- “Scale of SQL data types” on page 510
- “Length of SQL data types” on page 510
- “Display size of SQL data types” on page 511

Precision of SQL data types

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 nonnumeric column or parameter generally refers to the maximum length or the defined length of the column or parameter. Table 234 defines the precision for each SQL data type.

Table 234. Precision of SQL data types

fSqlType	Precision
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined number of characters for the column or parameter. For example, the precision of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length, in characters, of the column or parameter.[1]
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[2]	5
SQL_INTEGER[2]	10
SQL_FLOAT[2]	15
SQL_REAL[2]	7
SQL_ROWID	40
SQL_DOUBLE[2]	15
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length, in characters, 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, in characters, of the column or parameter.
SQL_TYPE_DATE[2]	10 (the number of characters in the yyyy-mm-dd format).
SQL_TYPE_TIME[2]	8 (the number of characters in the hh:mm:ss format).

Table 234. Precision of SQL data types (continued)

fSqlType	Precision
SQL_TYPE_TIMESTAMP	The number of characters in the "yyyy-mm-dd hh:mm:ss[.fff[fff]]" or "yyyy-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 26 (the number of characters in the "yyyy-mm-dd hh:mm:ss.ffffff" format). The maximum for fractional seconds is 6 digits.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length, in characters, of the column or parameter. For example, the precision of a column defined as GRAPHIC(10) is 10.
SQL_LONGVARGRAPHIC	The maximum length, in characters, of the column or parameter.

Notes:

| 1. When defining the precision of a parameter of this data type with SQLBindParameter(), *cbColDef* should be set to the total length in bytes of the data, not the precision as defined in this table.

| 2. The *cbColDef* argument of SQLBindParameter() is ignored for this data type.

Scale of SQL data types

The scale of a numeric column or parameter refers to the maximum number of digits to the right of the decimal point. For approximate floating point number columns or parameters, the scale is undefined, because the number of digits to the right of the decimal place is not fixed. Table 235 defines the scale for each SQL data type.

Table 235. Scale of SQL data types

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	0
SQL_REAL SQL_FLOAT SQL_DOUBLE	Not applicable.
SQL_ROWID	Not applicable.
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Not applicable.
SQL_TYPE_DATE SQL_TYPE_TIME	Not applicable.
SQL_TYPE_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. The maximum for fractional seconds is 6 digits.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC SQL_DBCLOB	Not applicable.

Length of SQL data types

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 nul-termination character. Note that the length of a column can 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. Table 236 defines the length for each SQL data type.

Table 236. Length of SQL data types

fSqlType	Length
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length, in bytes, of the column. For example, the length of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length, in bytes, of the column.
SQL_DECIMAL SQL_NUMERIC	The maximum number of digits plus two bytes. Because 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 bytes
SQL_INTEGER	4 bytes
SQL_REAL	4 bytes
SQL_ROWID	40 bytes
SQL_FLOAT	8 bytes
SQL_DOUBLE	8 bytes
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length, in bytes, of the column. For example, the length of a column defined as CHAR(10) FOR BIT DATA is 10.
SQL_LONGVARBINARY	The maximum length, in bytes, of the column.
SQL_TYPE_DATE SQL_TYPE_TIME	6 bytes (the size of the DATE_STRUCT or TIME_STRUCT structure).
SQL_TYPE_TIMESTAMP	16 bytes (the size of the TIMESTAMP_STRUCT structure).
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column times 2 bytes. For example, the length of a column defined as GRAPHIC(10) is 20.
SQL_LONGVARGRAPHIC	The maximum length of the column times 2 bytes.

Display size of SQL data types

The display size of a column is the maximum number of bytes that are needed to display data in character form. Table 237 defines the display size for each SQL data type.

Table 237. Display size of SQL data types

fSqlType	Display size
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length, in bytes, of the column. For example, the display size of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length, in bytes, of the column.
SQL_DECIMAL SQL_NUMERIC	The precision of the column plus two bytes (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 bytes (a sign and 5 digits).
SQL_INTEGER	11 bytes (a sign and 10 digits).
SQL_REAL	13 bytes (a sign, 7 digits, a decimal point, the letter E, a sign, and 2 digits).
SQL_ROWID	40 bytes
SQL_FLOAT SQL_DOUBLE	22 bytes (a sign, 15 digits, a decimal point, the letter E, a sign, and 3 digits).

Table 237. Display size of SQL data types (continued)

fSqlType	Display size
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length of the column times 2 bytes. (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 bytes.
SQL_TYPE_DATE	10 bytes (a date in the format yyyy-mm-dd).
SQL_TYPE_TIME	8 bytes (a time in the format hh:mm:ss).
SQL_TYPE_TIMESTAMP	19 bytes (if the scale of the timestamp is 0) or 20 bytes 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]]" or "yyyy-mm-dd.hh.mm.ss.fff[fff]]" format. For example, the display size of a column storing thousandths of a second is 23 bytes (the number of characters in "yyyy-mm-dd hh:mm:ss.ffffff"). The maximum for fractional seconds is 6 digits.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column or parameter times two bytes. For example, the display size of a column defined as GRAPHIC(10) is 20 bytes.
SQL_LONGVARGRAPHIC	The maximum length, in bytes, of the column or parameter.

Converting data from SQL to C data types

For each SQL data conversion type a table lists conversion information. Each column in these tables lists the following information:

- 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(), or SQLGetData().

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 **07**006 (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 **HY**C00 (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 nul-termination character. If *rgbValue* is a null pointer, SQLBindCol() or SQLGetData() returns SQLSTATE **HY**009 (Invalid argument value).

In the following tables:

Data length
: The total length, in bytes, of the data after it has been converted to the specified C data type (excluding the nul-termination character 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 238 shows information about converting character SQL data to C data. See "Converting data from SQL to C data types" on page 512 for a detailed description of each table column.

Table 238. Converting character SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Data length < cbValueMax	Data	Data length (in bytes)	**00**000[1]
	Data length >= cbValueMax	Truncated data	Data length (in bytes)	**01**004
SQL_C_BINARY	Data length <= cbValueMax	Data	Data length (in bytes)	**00**000[1]
	Data length > cbValueMax	Truncated data	Data length (in bytes)	**01**004
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_DOUBLE SQL_C_TINYINT SQL_C_BIT	Data converted without truncation[2]	Data	Size (in bytes) of the C data type	**00**000[1]
	Data converted with truncation, but without loss of significant digits[2]	Data	Size (in bytes) of the C data type	**01**004
	Conversion of data would result in loss of significant digits[2]	Untouched	Size (in bytes) of the C data type	**22**003
	Data is not a number[2]	Untouched	Size (in bytes) of the C data type	**22**005
SQL_C_TYPE_DATE	Data value is a valid date[2]	Data	6[3]	**00**000[1]
	Data value is not a valid date[2]	Untouched	6[3]	**22**008
SQL_C_TYPE_TIME	Data value is a valid time[2]	Data	6[3]	**00**000[1]
	Data value is not a valid time[2]	Untouched	6[3]	**22**008

Table 238. Converting character SQL data to C data (continued)

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_TYPE_TIMESTAMP	Data value is a valid timestamp[2]	Data	16[3]	**00**000[1]
	Data value is not a valid timestamp[2]	Untouched	16[3]	**22**008

Notes:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
2. 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.
3. This is the size of the corresponding C data type.

Converting graphic SQL data to C data

The graphic SQL data types are:

SQL_GRAPHIC
SQL_VARGRAPHIC
SQL_LONGVARGRAPHIC
SQL_DBCLOB

Table 239 shows information about converting graphic SQL data to C data. See "Converting data from SQL to C data types" on page 512 for a detailed description of each table column.

Table 239. 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 (in bytes)	**00**000[1]
	Number of double-byte characters * 2 <= cbValueMax	Truncated data, to the nearest even byte that is less than *cbValueMax*.	Length of data (in bytes)	**01**004
SQL_C_DBCHAR	Number of double-byte characters * 2 < cbValueMax	Data	Length of data (in bytes)	**00**000[1]
	Number of double-byte characters * 2 >= cbValueMax	Truncated *cbValueMax*. data, to the nearest even byte that is less than *cbValueMax*.	Length of data (in bytes)	**01**004

Note:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Converting numeric SQL data to C data

The numeric SQL data types are:

SQL_DECIMAL
SQL_NUMERIC
SQL_SMALLINT
SQL_INTEGER
SQL_REAL

SQL_FLOAT
SQL_DOUBLE

Table 240 shows information about converting numeric SQL data to C data. See “Converting data from SQL to C data types” on page 512 for a detailed description of each table column.

Table 240. Converting numeric SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Data length (in bytes)	**00**000[1]
	Number of significant digits < cbValueMax	Truncated data	Data length (in bytes)	**01**004
	Number of significant digits >= cbValueMax	Untouched	Data length (in bytes)	**22**003
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_DOUBLE SQL_C_TINYINT SQL_C_BIT	Data converted without truncation[2]	Data	Size (in bytes) of the C data type	**00**000[1]
	Data converted with truncation, but without loss of significant digits[2]	Truncated data	Size (in bytes) of the C data type	**01**004
	Conversion of data would result in loss of significant digits[2]	Untouched	Size (in bytes) of the C data type	**22**003

Notes:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
2. 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.

Converting binary SQL data to C data

The binary SQL data types are:
SQL_BINARY
SQL_VARBINARY
SQL_LONGVARBINARY
SQL_BLOB

Table 241 shows information about converting binary SQL data to C data. See “Converting data from SQL to C data types” on page 512 for a detailed description of each table column.

Table 241. Converting binary SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	(Data length) < cbValueMax	Data	Data length (in bytes)	N/A
	(Data length) >= cbValueMax	Truncated data	Data length (in bytes)	**01**004
SQL_C_BINARY	Data length <= cbValueMax	Data	Data length (in bytes)	N/A
	Data length > cbValueMax	Truncated data	Data length (in bytes)	**01**004

Converting date SQL data to C data

The date SQL data type is:
    SQL_TYPE_DATE

Table 242 shows information about converting date SQL data to C data. See "Converting data from SQL to C data types" on page 512 for a detailed description of each table column.

Table 242. Converting date SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 11	Data	10	**00**000[1]
	cbValueMax < 11	Untouched	10	**22**003
SQL_C_TYPE_DATE	None[2]	Data	6[4]	**00**000[1]
SQL_C_TYPE_TIMESTAMP	None[2]	Data[3]	16[4]	**00**000[1]
SQL_C_BINARY	Data length <= cbValueMax	Data	Data length (in bytes)	**00**000[1]
	Data length > cbValueMax	Untouched	Untouched	**22**003

Notes:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
2. 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.
3. The time fields of the TIMESTAMP_STRUCT structure are set to zero.
4. This is the size of the corresponding C data type.

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_TYPE_TIME

Table 243 shows information about converting time SQL data to C data. See "Converting data from SQL to C data types" on page 512 for a detailed description of each table column.

Table 243. Converting time SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 9	Data	8	**00**000[1]
	cbValueMax < 9	Untouched	8	**22**003
SQL_C_TYPE_TIME	None[2]	Data	6[3]	**00**000[1]
SQL_C_TYPE_TIMESTAMP	None[2]	Data[4]	16[3]	**00**000[1]

Table 243. Converting time SQL data to C data (continued)

fCType	Test	rgbValue	pcbValue	SQLSTATE

Notes:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
2. 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.
3. This is the size of the corresponding C data type.
4. 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.

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_TYPE_TIMESTAMP

Table 244 shows information about converting timestamp SQL data to C data. See “Converting data from SQL to C data types” on page 512 for a detailed description of each table column.

Table 244. Converting timestamp SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Data length (in bytes)	**00**000[1]
	19 <= cbValueMax <= Display size	Truncated data[2]	Data length (in bytes)	**01**004
	cbValueMax < 19	Untouched	Data length (in bytes)	**22**003
SQL_C_TYPE_DATE	None	Truncated data[3]	6[4]	**01**004
SQL_C_TYPE_TIME	None[5]	Truncated data[6]	6[4]	**01**004
SQL_C_TYPE_TIMESTAMP	None[5]	Data	16[4]	**00**000[1]
	Fractional seconds portion of timestamp is truncated.[5]	Data[2]	16	**01**004

Notes:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
2. The fractional seconds of the timestamp are truncated.
3. The time portion of the timestamp is deleted.
4. This is the size of the corresponding C data type.
5. 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.
6. The date portion of the timestamp is deleted.

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).

Converting row ID SQL data to C data

The row ID SQL data type is:
SQL_ROWID

Table 245 shows information about converting row ID SQL data to C data. See "Converting data from SQL to C data types" on page 512 for a detailed description of each table column.

Table 245. Converting row ID SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Data length < = cbValueMax	Data	Data length (in bytes)	**00**000
	Data length > cbValueMax	Truncated data	Data length (in bytes)	**01**004

SQL to C data conversion examples

Table 246 shows example SQL to C data conversions and the SQLSTATE associated with these conversions.

Table 246. SQL to C data conversion examples

SQL data type	SQL data value	C data type	cbValueMax	rgbValue	SQLSTATE
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0[1]	**00**000[2]
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0 [1]	**01**004
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0 [1]	**00**000[2]
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0 [1]	**01**004
SQL_DECIMAL	1234.56	SQL_C_CHAR	4	---	**22**003
SQL_DECIMAL	1234.56	SQL_C_FLOAT	Ignored	1234.56	**00**000[2]
SQL_DECIMAL	1234.56	SQL_C_SHORT	Ignored	1234	**01**004
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0 [1]	**00**000[2]
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	10	---	**22**003
SQL_TYPE_DATE	1992-12-31	SQL_C_TYPE_TIMESTAMP	Ignored	1992,12,31, 0,0,0,0[3]	**00**000[2]
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 [1]	**00**000[2]
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 [1]	**01**004
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	---	**22**003

Notes:

1. ″\0″ represents a nul-termination character.
2. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
3. The numbers in this list are the numbers stored in the fields of the TIMESTAMP_STRUCT structure.

Converting data from C to SQL data types

For each C data conversion type a table lists conversion information. Each column in these tables lists the following information:

- The first column of the table lists the legal input values of the *fSqlType* argument in SQLBindParameter().
- The second column lists the outcomes of a test, often using the length, in bytes, of the parameter data as specified in the *pcbValue* argument in SQLBindParameter(), 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().

  Important: Data is sent to the data source only if the SQLSTATE is **00**000 (success).

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

If the *fSqlType* argument in SQLBindParameter() contains a value not shown in the table for a given C data type, SQLSTATE **07**006 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() returns SQLSTATE **HY**C00 (Driver not capable).

If the *rgbValue* and *pcbValue* arguments specified in SQLBindParameter() are both null pointers, that function returns SQLSTATE **HY**009 (Invalid argument value).

Data length
: The total length in bytes of the data after it has been converted to the specified SQL data type (excluding the nul-termination character 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 nul-termination character.

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 247 on page 520 shows information about converting character C data to SQL data. See “Converting data from C to SQL data types” for a detailed description of each table column.

Table 247. Converting character C data to SQL data

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Data length <= Column length	**00**000[1]
	Data length > Column length	**01**004
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_REAL SQL_FLOAT SQL_DOUBLE	Data converted without truncation	**00**000[1]
	Data converted with truncation, but without loss of significant digits	**01**004
	Conversion of data would result in loss of significant digits	**22**003
	Data value is not a numeric value	**22**005
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	(Data length) < Column length	N/A
	(Data length) >= Column length	**01**004
	Data value is not a hexadecimal value	**22**005
SQL_ROWID	Data length <= Column length	**00**000[1]
	Data length > Column length	**01**004
SQL_TYPE_DATE	Data value is a valid date	**00**000[1]
	Data value is not a valid date	**22**008
SQL_TYPE_TIME	Data value is a valid time	**00**000[1]
	Data value is not a valid time	**22**008
	Data value is a valid timestamp; time portion is nonzero	**01**004
SQL_TYPE_TIMESTAMP	Data value is a valid timestamp Data value is a valid timestamp; fractional seconds portion is nonzero	**00**000[1] or **01**004
	Data value is not a valid timestamp Data value is a valid timestamp; fractional seconds portion is nonzero	**22**008 or **01**004
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC SQL_DBCLOB	Data length / 2 <= Column length	**00**000[1]
	Data length / 2 < Column length	**01**004

Note:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), 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 248 on page 521 shows information about converting numeric C data to SQL data. See "Converting data from C to SQL data types" on page 519 for a detailed description of each table column.

Table 248. 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	**00**000[1]
	Data converted with truncation, but without loss of significant digits	**01**004
	Conversion of data would result in loss of significant digits	**22**003
SQL_CHAR SQL_VARCHAR	Data converted without truncation.	**00**000[1]
	Conversion of data would result in loss of significant digits.	**22**003

Note:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Converting binary C data to SQL data

The binary C data type is:
SQL_C_BINARY

Table 249 shows information about converting binary C data to SQL data. See "Converting data from C to SQL data types" on page 519 for a detailed description of each table column.

Table 249. Converting binary C data to SQL data

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Data length <= Column length	N/A
	Data length > Column length	**01**004
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Data length <= Column length	N/A
	Data length > Column length	**01**004

Converting double-byte character C data to SQL data

The double-byte C data type is:
SQL_C_DBCHAR

Table 250 shows information about converting double-byte character C data to SQL data. See "Converting data from C to SQL data types" on page 519 for a detailed description of each table column.

Table 250. Converting double-byte character C data to SQL data

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Data length <= Column length x 2	N/A
	Data length > Column length x 2	**01**004
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Data length <= Column length x 2	N/A
	Data length > Column length x 2	**01**004

Converting date C data to SQL data

The date C data type is:
SQL_C_TYPE_DATE

Table 251 shows information about converting date C data to SQL data. See "Converting data from C to SQL data types" on page 519 for a detailed description of each table column.

Table 251. Converting date C data to SQL data

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 10	**00**000[1]
	Column length < 10	**22**003
SQL_TYPE_DATE	Data value is a valid date	**00**000[1]
	Data value is not a valid date	**22**008
SQL_TYPE_TIMESTAMP[2]	Data value is a valid date	**00**000[1]
	Data value is not a valid date	**22**008

Notes:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
2. The time component of TIMESTAMP is set to zero.

Converting time C data to SQL data

The time C data type is:
SQL_C_TYPE_TIME

Table 252 shows information about converting time C data to SQL data. See "Converting data from C to SQL data types" on page 519 for a detailed description of each table column.

Table 252. Converting time C data to SQL data

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 8	**00**000[1]
	Column length < 8	**22**003
SQL_TYPE_TIME	Data value is a valid time	**00**000[1]
	Data value is not a valid time	**22**008
SQL_TYPE_TIMESTAMP[2]	Data value is a valid time	**00**000[1]
	Data value is not a valid time	**22**008

Notes:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
2. 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_TYPE_TIMESTAMP

Table 253 shows information about converting timestamp C data to SQL data. See "Converting data from C to SQL data types" on page 519 for a detailed description of each table column.

Table 253. Converting timestamp C data to SQL data

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= Display size	**00**000[1]
	19 <= Column length < Display size[2]	**01**004
	Column length < 19	**22**003
SQL_TYPE_DATE	Data value is a valid date[3]	**01**004
	Data value is not a valid date	**22**008
SQL_TYPE_TIME	Data value is a valid time[4]	**01**004
	Data value is not a valid time Fractional seconds fields are nonzero	**22**008 **01**004
SQL_TYPE_TIMESTAMP	Data value is a valid timestamp	**00**000[1]
	Data value is not a valid timestamp	**22**008

Notes:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
2. The fractional seconds of the timestamp are truncated.
3. The time portion of the timestamp is deleted.
4. The date portion of the timestamp is deleted.

C to SQL data conversion examples

Table 254 shows example C to SQL data conversions and the SQLSTATE associated with these conversions.

Table 254. C to SQL data conversion examples

C data type	C data Value	SQL data type	Column length	SQL data value	SQLSTATE
SQL_C_CHAR	abcdef\0	SQL_CHAR	6	abcdef	**00**000[1]
SQL_C_CHAR	abcdef\0	SQL_CHAR	5	abcde	**01**004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6	1234.56	**00**000[1]
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	5	1234.5	**01**004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	3	---	**22**003
SQL_C_FLOAT	1234.56	SQL_FLOAT	Not applicable	1234.56	**00**000[1]
SQL_C_FLOAT	1234.56	SQL_INTEGER	Not applicable	1234	**01**004

Note:

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Appendix E. Deprecated functions

This appendix explains DB2 UDB for z/OS's support of the ODBC 3.0 standard. DB2 ODBC introduces ODBC 3.0 support in Version 6.

Mapping deprecated functions

The ODBC 3.0 functions replace, or *deprecate*, many existing ODBC 2.0 functions. The DB2 ODBC driver continues to support all of the deprecated functions.

Recommendation: Begin using ODBC 3.0 functional replacements to maintain optimum portability.

Table 255 lists the ODBC 2.0 deprecated functions and the ODBC 3.0 replacement functions.

Table 255. ODBC 2.0 deprecated functions

ODBC 2.0 deprecated function	Purpose	ODBC 3.0 replacement function
SQLAllocConnect()	Obtains an connection handle.	SQLAllocHandle() with *HandleType*=SQL_HANDLE_DBC
SQLAllocEnv()	Obtains an environment handle.	SQLAllocHandle() with *HandleType*=SQL_HANDLE_ENV
SQLAllocStmt()	Obtains an statement handle.	SQLAllocHandle() with *HandleType*=SQL_HANDLE_STMT
SQLColAttributes()	Gets column attributes.	SQLColAttribute()
SQLError()	Returns additional diagnostic information (multiple fields of the diagnostic data structure).	SQLGetDiagRec()
SQLFreeConnect()	Frees connection handle.	SQLFreeHandle() with *HandleType*=SQL_HANDLE_DBC
SQLFreeEnv()	Frees environment handle.	SQLFreeHandle() with *HandleType*=SQL_HANDLE_ENV
SQLFreeStmt() with *fOption*=SQL_DROP	Frees a statement handle.	SQLFreeHandle() with *HandleType*= SQL_HANDLE_STMT
SQLGetConnectOption()	Returns a value of a connection attribute.	SQLGetConnectAttr()
SQLGetStmtOption()	Returns a value of a statement attribute.	SQLGetStmtAttr()
SQLSetConnectOption()	Sets a value of a connection attribute.	SQLSetConnectAttr()
SQLSetParam()	Binds a parameter marker to an application variable.	SQLBindParameter()
SQLSetStmtOption()	Sets a value of a statement attribute.	SQLSetStmtAttr()
SQLTransact()	Commits or rolls back a transaction.	SQLEndTran()

Changes to SQLGetInfo() information types

Values of the *InfoType* arguments for SQLGetInfo() arguments were renamed in ODBC 3.0. Table 120 on page 255 lists the ODBC 2.0 and ODBC 3.0 argument names.

Changes to SQLSetConnectAttr() attributes

The renamed ODBC 3.0 attribute values support all existing ODBC 2.0 attributes. You can specify either the ODBC 2.0 or the ODBC 3.0 attribute value; the ODBC driver supports both. Table 256 matches ODBC 2.0 and ODBC 3.0 values.

Table 256. SQLSetConnectAttr() attribute value mapping

ODBC 2.0 attribute	ODBC 3.0 attribute
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_MAXCONN	SQL_ATTR_MAXCONN
SQL_PARAMOPT_ATOMIC	SQL_ATTR_PARAMOPT_ATOMIC
SQL_SYNC_POINT	SQL_ATTR_SYNC_POINT
SQL_TXN_ISOLATION	SQL_ATTR_TXN_ISOLATION

Changes to SQLSetEnvAttr() attributes

Table 257 lists the SQLSetEnvAttr() attribute values renamed in ODBC 3.0. The ODBC 3.0 attributes support all of the existing ODBC 2.0 attributes. You can specify either the ODBC 2.0 or the ODBC 3.0 attribute value; the ODBC driver supports both.

Table 257. SQLSetEnvAttr() attribute value mapping

ODBC 2.0 attribute	ODBC 3.0 attribute
SQL_CONNECTTYPE	SQL_ATTR_CONNECTTYPE
SQL_MAXCONN	SQL_ATTR_MAXCONN
SQL_OUTPUT_NTS	SQL_ATTR_OUTPUT_NTS

Changes to SQLSetStmtAttr() attributes

Table 258 lists the SQLSetStmtAttr() attribute values renamed in ODBC 3.0. The ODBC 3.0 attributes support all of the existing ODBC 2.0 attributes. You can specify either the ODBC 2.0 or the ODBC 3.0 attribute value; the ODBC driver supports both.

Table 258. SQLSetStmtAttr() attribute value mapping

ODBC 2.0 attribute	ODBC 3.0 attribute
SQL_BIND_TYPE	SQL_ATTR_BIND_TYPE or SQL_ATTR_ROW_BIND_TYPE
SQL_CLOSE_BEHAVIOR	SQL_ATTR_CLOSE_BEHAVIOR
SQL_CONCURRENCY	SQL_ATTR_CONCURRENCY

Table 258. SQLSetStmtAttr() attribute value mapping (continued)

ODBC 2.0 attribute	ODBC 3.0 attribute
SQL_CURSOR_HOLD	SQL_ATTR_CURSOR_HOLD
SQL_CURSOR_TYPE	SQL_ATTR_CURSOR_TYPE
SQL_MAX_LENGTH	SQL_ATTR_MAX_LENGTH
SQL_MAX_ROWS	SQL_ATTR_MAX_ROWS
SQL_NODESCRIBE	SQL_ATTR_NODESCRIBE
SQL_NOSCAN	SQL_ATTR_NOSCAN
SQL_RETRIEVE_DATA	SQL_ATTR_RETRIEVE_DATA
SQL_ROWSET_SIZE	SQL_ATTR_ROWSET_SIZE or SQL_ATTR_ROW_ARRAY_SIZE
SQL_STMTTXN_ISOLATION or SQL_TXN_ISOLATION	SQL_ATTR_STMTTXN_ISOLATION or SQL_ATTR_TXN_ISOLATION

ODBC 3.0 driver behavior

Behavioral changes refer to functionality that varies depending on the version of ODBC in use. The ODBC 2.0 and ODBC 3.0 drivers behave according to the setting of the SQL_ATTR_ODBC_VERSION environment attribute.

The SQL_ATTR_ODBC_VERSION environment attribute controls whether the DB2 ODBC 3.0 driver driver exhibits ODBC 2.0 or ODBC 3.0 behavior. This value is implicitly set by the ODBC driver by application calls to the ODBC 3.0 function SQLAllocHandle() or the ODBC 2.0 function SQLAllocEnv(). The application can explicitly set by calls to SQLSetEnvAttr().

- ODBC 3.0 applications first call SQLAllocHandle() to get the environmental handle. The DB2 ODBC 3.0 driver implicitly sets SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC3. This setting ensures that ODBC 3.0 applications get ODBC 3.0 behavior.

  An ODBC 3.0 application should not invoke SQLAllocHandle() and then call SQLAllocEnv(). Doing so implicitly resets the application to ODBC 2.0 behavior. To avoid resetting an application to ODBC 2.0 behavior, ODBC 3.0 applications should always use SQLAllocHandle() to manage environment handles.
- ODBC 2.0 applications first call SQLAllocEnv() to get the environmental handle. The DB2 ODBC 2.0 driver implicitly sets SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC2. This setting ensures that ODBC 2.0 applications get ODBC 2.0 behavior.

An application can verify the ODBC version setting by calling SQLGetEnvAttr() for attribute SQL_ATTR_ODBC_VERSION. An application can explicitly set the ODBC version setting by calling SQLSetEnvAttr() for attribute SQL_ATTR_ODBC_VERSION.

Forward compatibility does not affect ODBC 2.0 applications that were compiled using the previous DB2 ODBC 2.0 driver header files, or ODBC 2.0 applications that are recompiled using the new ODBC 3.0 header files. These applications can continue executing as ODBC 2.0 applications on the DB2 ODBC 3.0 driver. These ODBC 2.0 applications need not call SQLSetEnvAttr(). As stated above, when the existing ODBC 2.0 application calls SQLAllocEnv() (ODBC 2.0 API to allocate environment handle), the DB2 ODBC 3.0 driver will implicitly set

SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC2. This will ensure ODBC 2.0 driver behavior when using the DB2 ODBC 3.0 driver.

SQLSTATE mappings

Several SQLSTATEs will differ when you call SQLGetDiagRec() or SQLError() under an ODBC 3.0 driver:

- **HY**xxx SQLSTATEs replace **S1**xxx SQLSTATEs
- **42**Sxx SQLSTATEs replace **S0**0xx SQLSTATEs
- Several SQLSTATEs are redefined

When an ODBC 2.0 application is upgraded to ODBC 3.0, the application must be changed to expect the ODBC 3.0 SQLSTATEs. An ODBC 3.0 application can set the environment attribute SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC2 to enable the DB2 ODBC 3.0 driver to return the ODBC 2.0 SQLSTATEs.

All deprecated functions continue to return ODBC 2.0 SQLSTATEs regardless of which environment attributes are set.

Table 259 lists ODBC 2.0 to ODBC 3.0 SQLSTATE mappings.

Table 259. ODBC 2.0 to ODBC 3.0 SQLSTATE mappings

ODBC 2.0 SQLSTATE	ODBC 3.0 SQLSTATE
22003	**HY**019
22007	**22**008
22005	**22**018
37000	**42**000
S0001	**42**S01
S0002	**42**S02
S0011	**42**S11
S0012	**42**S12
S0021	**42**S21
S0022	**42**S22
S0023	**42**S23
S1000	**HY**000
S1001	**HY**001
S1002	**HY**002
S1003	**HY**003
S1004	**HY**004
S1009	**HY**009 or **HY**024 **S1**009 is mapped to **HY**009 for invalid use of null pointers; **S1**009 is mapped to **HY**024 for invalid attribute values.
S1010	**HY**010
S1011	**HY**011
S1012	**HY**012
S1013	**HY**013
S1014	**HY**014

Table 259. ODBC 2.0 to ODBC 3.0 SQLSTATE mappings (continued)

ODBC 2.0 SQLSTATE	ODBC 3.0 SQLSTATE
S1015	**HY**015
S1019	**HY**019
S1090	**HY**090
S1091	**HY**091
S1092	**HY**092
S1093	**HY**093
S1096	**HY**096
S1097	**HY**097
S1098	**HY**098
S1099	**HY**099
S1100	**HY**100
S1101	**HY**101
S1103	**HY**103
S1104	**HY**104
S1105	**HY**105
S1106	**HY**106
S1107	**HY**107
S1110	**HY**110
S1501	**HY**501
S1506	**HY**506
S1C00	**HY**C00

Changes to datetime data types

In ODBC 3.0, the identifiers for date, time, and timestamp have changed. The `#define` directives in the include file sqlcli1.h are added for the values defined in Table 260 for SQL type mappings and Table 261 on page 530 for C type mappings.

- For input, either ODBC 2.0 or ODBC 3.0 datetime values can be used with the DB2 ODBC 3.0 driver.
- On output, the DB2 ODBC 3.0 driver determines the appropriate value to return based on the setting of the SQL_ATTR_ODBC_VERSION environment attribute.
  - If SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC2, the output datetime values are the ODBC 2.0 values.
  - If SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC3, the output datetime values are the ODBC 3.0 values.

Table 260. Datetime data type mappings: SQL type identifiers

ODBC 2.0	ODBC 3.0
SQL_DATE(9)	SQL_TYPE_DATE(91)
SQL_TIME(10)	SQL_TYPE_TIME(92)
SQL_TIMESTAMP(11)	SQL_TYPE_TIMETAMP(93)

Changes to datetime data types

Table 261. Datetime data type mappings: C type identifiers

ODBC 2.0	**ODBC 3.0**
SQL_C_DATE(9)	SQL_C_TYPE_DATE(91)
SQL_C_TIME(10)	SQL_C_TYPE_TIME(92)
SQL_C_TIMESTAMP(11)	SQL_C_TYPE_TIMESTAMP(93)

The datetime data type changes affect the following functions:
- SQLBindCol()
- SQLBindParameter()
- SQLColAttribute()
- SQLColumns()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLGetData()
- SQLGetTypeInfo()
- SQLProcedureColumns()
- SQLStatistics()
- SQLSpecialColumns()

Appendix F. Example DB2 ODBC code

This appendix provides DB2 ODBC samples:

- "DSN8O3VP sample application" shows the sample verification program DSN8O3VP available online in the DSN810.SDSNSAMP data set. You can use this sample to verify that your DB2 ODBC 3.0 installation is correct. See "Application preparation and execution steps" on page 46 for more information about online sample applications.
- In "Client application calling a DB2 ODBC stored procedure" on page 537, a client application (APD29) calls a DB2 ODBC stored procedure (SPD29). It includes very fundamental processing of query result sets (a query cursor opened in a stored procedure and return to client for fetching). For completeness, the CREATE TABLE, data INSERTs and CREATE PROCEDURE definition is provided.

DSN8O3VP sample application

"DSN8O3VP sample application" presents the code that the sample program DSN810.SDSNSAMP(DSN8O3VP) contains. The DSN8O3VP program validates the installation of DB2 ODBC.

```
 /*********************************************************************/
 /*  DB2 ODBC 3.0 installation certification test to validate        */
 /*  installation.                                                   */
 /*                                                                  */
 /*  DSNTEJ8 is sample JCL to that can be used to run this           */
 /*  application.                                                    */
 /*********************************************************************/
 /******************************************************************/
 /* Include the 'C' include files                                  */
 /******************************************************************/
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
   #include "sqlcli1.h"
   /******************************************************************/
   /* Variables                                                      */
   /******************************************************************/
#ifndef NULL
#define NULL   0
#endif
    SQLHENV henv = SQL_NULL_HENV;
    SQLHDBC hdbc = SQL_NULL_HDBC;
    SQLHDBC hstmt= SQL_NULL_HSTMT;
    SQLRETURN rc = SQL_SUCCESS;
    SQLINTEGER      id;
    SQLCHAR         name[51]
    SQLINTEGER      namelen, intlen, colcount;
    struct sqlca    sqlca;
    SQLCHAR   server[18]
    SQLCHAR   uid[30]
    SQLCHAR   pwd[30]
    SQLCHAR   sqlstmt[500]
SQLRETURN check_error(SQLSMALLINT,SQLHANDLE,SQLRETURN,int,char *);
SQLRETURN print_error(SQLSMALLINT,SQLHANDLE,SQLRETURN,int,char *);
SQLRETURN prt_sqlca(void);
#define CHECK_HANDLE( htype, hndl, rc ) if ( rc != SQL_SUCCESS ) \
   {check_error(htype,hndl,rc,__LINE__,__FILE__);goto dberror;}
```

Figure 63. DSN8O3VP sample application (Part 1 of 6)

```
   /******************************************************************/
   /* Main Program                                                   */
   /******************************************************************/
int main()
{
 printf("DSN8O3VP INITIALIZATION\n");
 printf("DSN8O3VP SQLAllocHandle-Environment\n");
 henv=0;
 rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
 CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc );
 printf("DSN8O3VP-henv=%i\n",henv);
 printf("DSN8O3VP SQLAllocHandle-Environment successful\n");
 printf("DSN8O3VP SQLAllocHandle-Connection\n");
 hdbc=0;
 rc=SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc);
 CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
 printf("DSN8O3VP-hdbc=%i\n",hdbc);
 printf("DSN8O3VP SQLAllocHandle-Connection successful\n");
 printf("DSN8O3VP SQLConnect\n");
 strcpy((char *)uid,"");
 strcpy((char *)pwd,"");
 strcpy((char *)server,"ignore");
 /* sample is NULL connect to default datasource */
 rc=SQLConnect(hdbc,NULL,0,NULL,0,NULL,0);
 CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
 printf("DSN8O3VP successfully issued a SQLconnect\n");
 printf("DSN8O3VP SQLAllocHandle-Statement\n");
 hstmt=0;
 rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
 printf("DSN8O3VP hstmt=%i\n",hstmt);
 printf("DSN8O3VP SQLAllocHandle-Statement successful\n");
 printf("DSN8O3VP SQLExecDirect\n");
 strcpy((char *)sqlstmt,"SELECT * FROM SYSIBM.SYSDUMMY1");
 printf("DSN8O3VP sqlstmt=%s\r",sqlstmt);
 rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
 printf("DSN8O3VP successfully issued a SQLExecDirect\n");
 /* sample fetch without looking at values */
 printf("DSN8O3VP SQLFetch\n");
 rc=SQLFetch(hstmt);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
 printf("DSN8O3VP successfully issued a SQLFetch\n");
 printf("DSN8O3VP SQLEndTran-Commit\n");
 rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
 printf("DSN8O3VP SQLEndTran-Commit successful\n");
 printf("DSN8O3VP SQLFreeHandle-Statement\n");
 rc=SQLFreeHandle(SQL_HANDLE_STMT,hstmt);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
 hstmt=0;
 printf("DSN8O3VP SQLFreeHandle-Statement successful\n");
```

Figure 63. DSN8O3VP sample application (Part 2 of 6)

```
 /******** SQLDisconnect ******************************************/
 printf("DSN8O3VP SQLDisconnect\n");
 rc=SQLDisconnect(hdbc);
 CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
 printf("DSN8O3VP successfully issued a SQLDisconnect\n");
 /******** SQLFreeConnect *****************************************/
 printf("DSN8O3VP SQLFreeHandle-Connection\n");
 rc=SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
 CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
 hdbc=0;
 printf("DSN8O3VP SQLFreeHandle-Connection successful\n");
 /******** SQLFreeEnv *********************************************/
 printf("DSN8O3VP SQLFreeHandle-Environment\n");
 rc=SQLFreeHandle(SQL_HANDLE_ENV,henv);
 CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc );
 henv=0;
 printf("DSN8O3VP SQLFreeHandle-Environment successful\n");
 pgmend:
 printf("DSN8O3VP pgmend: Ending sample\n");
 if (rc==0)
  printf("DSN8O3VP Execution was SUCCESSFUL\n");
 else
  {
  printf("DSN8O3VP***************************\n");
  printf("DSN8O3VP Execution FAILED\n");
  printf("DSN8O3VP rc = %i\n", rc  );
  printf("DSN8O3VP ***************************\n");
  }
 return(rc);
 dberror:
 printf("DSN8O3VP dberror: entry dberror rtn\n");
 printf("DSN8O3VP dberror: rc=%d\n",rc);
 printf("DSN8O3VP dberror: environment cleanup attempt\n");
 printf("DSN8O3VP dberror: cleanup SQLFreeEnv\n");
 rc=SQLFreeEnv(henv);
 printf("DSN8O3VP dberror: cleanup SQLFreeEnv rc =%d\n",rc);
 rc=12;
 printf("DSN8O3VP dberror: setting error rc=%d\n",rc);
 goto pgmend;
}  /*END MAIN*/
```

Figure 63. DSN8O3VP sample application (Part 3 of 6)

```
/******************************************************************/
/* check_error                                                    */
/******************************************************************/
/*   RETCODE values   from sqlcli.h                               */
/*#define  SQL_SUCCESS            0                               */
/*#define  SQL_SUCCESS_WITH_INFO  1                               */
/*#define  SQL_NO_DATA_FOUND      100                             */
/*#define  SQL_NEED_DATA          99                              */
/*#define  SQL_NO_DATA            SQL_NO_DATA_FOUND               */
/*#define  SQL_STILL_EXECUTING    2       not currently returned */
/*#define  SQL_ERROR              -1                              */
/*#define  SQL_INVALID_HANDLE     -2                              */
/******************************************************************/
SQLRETURN check_error( SQLSMALLINT htype, /* A handle type */
                       SQLHANDLE   hndl,  /* A handle */
                       SQLRETURN   frc,   /* Return code */
                       int         line,  /* Line error issued */
                       char *      file   /* file error issued */
                     ) {
    SQLCHAR         cli_sqlstate[SQL_SQLSTATE_SIZE + 1];
    SQLINTEGER      cli_sqlcode;
    SQLSMALLINT     length;
    printf("DSN8O3VP entry check_error rtn\n");
    switch (frc) {
    case SQL_SUCCESS:
      break;
    case SQL_INVALID_HANDLE:
      printf("DSN8O3VP check_error> SQL_INVALID HANDLE \n");
      break;
    case SQL_ERROR:
      printf("DSN8O3VP check_error> SQL_ERROR\n");
      break;
    case SQL_SUCCESS_WITH_INFO:
      printf("DSN8O3VP check_error>  SQL_SUCCESS_WITH_INFO\n");
      break;
    case SQL_NO_DATA_FOUND:
      printf("DSN8O3VP check_error> SQL_NO_DATA_FOUND\n");
      break;
    default:
      printf("DSN8O3VP check_error> Received rc from api rc=%i\n",frc);
      break;
    } /*end switch*/
    print_error(htype,hndl,frc,line,file);
    printf("DSN8O3VP SQLGetSQLCA\n");
    rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    if( rc == SQL_SUCCESS )
      prt_sqlca();
    else
      printf("DSN8O3VP check_error SQLGetSQLCA failed rc=%i\n",rc);
    printf("DSN8O3VP exit check_error rtn\n");
    return (frc);
}  /* end check_error */
```

Figure 63. DSN8O3VP sample application (Part 4 of 6)

```
/******************************************************************/
/* print_error                                                    */
/* calls SQLGetDiagRec() displays SQLSTATE and message            */
/******************************************************************/
SQLRETURN print_error( SQLSMALLINT htype, /* A handle type */
                       SQLHANDLE   hndl,  /* A handle */
                       SQLRETURN   frc,   /* Return code */
                       int         line,  /* error from line */
                       char *      file   /* error from file */
                     ) {
    SQLCHAR     buffer[SQL_MAX_MESSAGE_LENGTH + 1] ;
    SQLCHAR     sqlstate[SQL_SQLSTATE_SIZE + 1] ;
    SQLINTEGER  sqlcode ;
    SQLSMALLINT length, i ;
    SQLRETURN   prc;
    printf("DSN8O3VP entry print_error rtn\n");
    printf("DSN8O3VP rc=%d reported from file:%s,line:%d ---\n",
            frc,
            file,
            line
          ) ;
    i = 1 ;
    while ( SQLGetDiagRec( htype,
                           hndl,
                           i,
                           sqlstate,
                           &sqlcode,
                           buffer,
                           SQL_MAX_MESSAGE_LENGTH + 1,
                           &length
                         ) == SQL_SUCCESS ) {
       printf( "DSN8O3VP SQLSTATE: %s\n", sqlstate ) ;
       printf( "DSN8O3VP Native Error Code: %ld\n", sqlcode ) ;
       printf( "DSN8O3VP buffer: %s \n", buffer ) ;
       i++ ;
    }
    printf( ">--------------------------------------------------\n" ) ;
    printf("DSN8O3VP exit print_error rtn\n");
    return( SQL_ERROR ) ;
}  /* end print_error */
```

Figure 63. DSN8O3VP sample application (Part 5 of 6)

```
/******************************************************************/
/* prt_sqlca                                                      */
/******************************************************************/
SQLRETURN
  prt_sqlca()
  {
    int i;
    printf("DSN8O3VP entry prt_sqlca rtn\n");
    printf("\r\rDSN8O3VP*** Printing the SQLCA:\r");
    printf("\nDSN8O3VP SQLCAID .... %s",sqlca.sqlcaid);
    printf("\nDSN8O3VP SQLCABC .... %d",sqlca.sqlcabc);
    printf("\nDSN8O3VP SQLCODE .... %d",sqlca.sqlcode);
    printf("\nDSN8O3VP SQLERRML ... %d",sqlca.sqlerrml);
    printf("\nDSN8O3VP SQLERRMC ... %s",sqlca.sqlerrmc);
    printf("\nDSN8O3VP SQLERRP  ... %s",sqlca.sqlerrp);
    for (i = 0; i < 6; i++)
      printf("\nDSN8O3VP SQLERRD%d ... %d",i+1,sqlca.sqlerrd??(i??));
    for (i = 0; i < 10; i++)
      printf("\nDSN8O3VP SQLWARN%d ... %c",i,sqlca.sqlwarn[i]);
    printf("\nDSN8O3VP SQLWARNA ... %c",sqlca.sqlwarn[10]);
    printf("\nDSN8O3VP SQLSTATE ... %s",sqlca.sqlstate);
    printf("\nDSN8O3VP exit prt_sqlca rtn\n");
    return(0);
  }  /* End of prt_sqlca */
/*******************************************************************/
/* END DSN8O3VP                                                    */
/*******************************************************************/
```

Figure 63. DSN8O3VP sample application (Part 6 of 6)

Client application calling a DB2 ODBC stored procedure

"Client application calling a DB2 ODBC stored procedure" presents a client application (APD29) that calls a DB2 ODBC stored procedure (SPD29) and processes query result sets (a query cursor opened in a stored procedure and return to client for fetching). The CREATE TABLE, data INSERT, and CREATE PROCEDURE statements are provided to define the DB2 objects and procedures that this example uses.

```
STEP 1. Create table

     printf("\nAPDDL SQLExecDirect  stmt=%d",__LINE__);
    strcpy((char *)sqlstmt,
 "CREATE TABLE TABLE2A (INT4 INTEGER,SMINT SMALLINT,FLOAT8 FLOAT");
    strcat((char *)sqlstmt,
 ",DEC312 DECIMAL(31,2),CHR10 CHARACTER(10),VCHR20 VARCHAR(20)");
    strcat((char *)sqlstmt,
 ",LVCHR LONG VARCHAR,CHRSB CHAR(10),CHRBIT CHAR(10) FOR BIT DATA");
    strcat((char *)sqlstmt,
 ",DDATE DATE,TTIME TIME,TSTMP TIMESTAMP)");
    printf("\nAPDDL sqlstmt=%s",sqlstmt);
    rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;

STEP 2. Insert 101 rows into table

    /* insert 100 rows into table2a */
    for (jx=1;jx<=100 ;jx++ ) {
      printf("\nAPDIN SQLExecDirect  stmt=%d",__LINE__);
      strcpy((char *)sqlstmt,"insert into table2a values(");
      sprintf((char *)sqlstmt+strlen((char *)sqlstmt),"%ld",jx);
      strcat((char *)sqlstmt,
 ",4,8.2E+30,1515151515151.51,'CHAR','VCHAR','LVCCHAR','SBCS'");
      strcat((char *)sqlstmt,
 ",'MIXED','01/01/1991','3:33 PM','1999-09-09-09.09.09.090909')");
      printf("\nAPDIN sqlstmt=%s",sqlstmt);
      rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
      if( rc != SQL_SUCCESS ) goto dberror;
    } /* endfor */

STEP 3. Define stored procedure with CREATE
PROCEDURE SQL statement

  CREATE PROCEDURE SPD29
  (INOUT INTEGER)
  PROGRAM TYPE MAIN
  EXTERNAL NAME SPD29
  COLLID DSNAOCLI
  LANGUAGE C
  RESULT SET 2
  MODIFIES SQL DATA
  PARAMETER STYLE GENERAL
  NO WLM ENVIRONMENT;
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 1 of 20)

STEP 4. Stored procedure

```
  /*START OF SPD29*****************************************************/
   /* PRAGMA TO CALL PLI SUBRTN CSPSUB TO ISSUE CONSOLE MSGS        */
   #pragma options (rent)
   #pragma runopts(plist(os))
   /******************************************************************/
   /* Include the 'C' include files                                  */
   /******************************************************************/
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
   #include "sqlcli1.h"
   #include <sqlca.h>
   #include <decimal.h>
   #include <wcstr.h>
   /******************************************************************/
   /* Variables for COMPARE routines                                 */
   /******************************************************************/
#ifndef NULL
#define NULL   0
#endif

    SQLHENV henv = SQL_NULL_HENV;
    SQLHDBC hdbc = SQL_NULL_HDBC;
    SQLHSTMT hstmt = SQL_NULL_HSTMT;
    SQLHSTMT hstmt2 = SQL_NULL_HSTMT;
    SQLRETURN rc = SQL_SUCCESS;
    SQLINTEGER      id;
    SQLCHAR         name[51];
    SQLINTEGER      namelen, intlen, colcount;
    SQLSMALLINT     scale;
    struct sqlca    sqlca;
    SQLCHAR         server[18];
    SQLCHAR         uid[30];
    SQLCHAR         pwd[30];
    SQLCHAR         sqlstmt[500];
    SQLCHAR         sqlstmt2[500];
    SQLSMALLINT     pcpar=0;
    SQLSMALLINT     pccol=0;
    SQLCHAR         cursor[19];
    SQLSMALLINT     cursor_len;

    SQLINTEGER    SPCODE;
    struct {
      SQLSMALLINT LEN;
      SQLCHAR     DATA_200¨; }         STMTSQL;

    SQLSMALLINT             H1SMINT;
    SQLINTEGER              H1INT4;
    SQLDOUBLE               H1FLOAT8;
    SQLDOUBLE               H1DEC312;
    SQLCHAR                 H1CHR10[11];
    SQLCHAR                 H1VCHR20[21];
    SQLCHAR                 H1LVCHR[21];
    SQLCHAR                 H1CHRSB[11];
    SQLCHAR                 H1CHRBIT[11];
    SQLCHAR                 H1DDATE[11];
    SQLCHAR                 H1TTIME[9];
    SQLCHAR                 H1TSTMP[27];
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 2 of 20)

```
SQLSMALLINT              I1SMINT;
SQLSMALLINT              I1INT4;
SQLSMALLINT              I1FLOAT8;
SQLSMALLINT              I1DEC312;
SQLSMALLINT              I1CHR10;
SQLSMALLINT              I1VCHR20;
SQLSMALLINT              I1LVCHR;
SQLSMALLINT              I1CHRSB;
SQLSMALLINT              I1CHRBIT;
SQLSMALLINT              I1DDATE;
SQLSMALLINT              I1TTIME;
SQLSMALLINT              I1TSTMP;

SQLINTEGER               LEN_H1SMINT;
SQLINTEGER               LEN_H1INT4;
SQLINTEGER               LEN_H1FLOAT8;
SQLINTEGER               LEN_H1DEC312;
SQLINTEGER               LEN_H1CHR10;
SQLINTEGER               LEN_H1VCHR20;
SQLINTEGER               LEN_H1LVCHR;
SQLINTEGER               LEN_H1CHRSB;
SQLINTEGER               LEN_H1CHRBIT;
SQLINTEGER               LEN_H1DDATE;
SQLINTEGER               LEN_H1TTIME;
SQLINTEGER               LEN_H1TSTMP;

SQLSMALLINT              H2SMINT;
SQLINTEGER               H2INT4;
SQLDOUBLE                H2FLOAT8;
SQLCHAR                  H2CHR10[11];
SQLCHAR                  H2VCHR20[21];
SQLCHAR                  H2LVCHR[21];
SQLCHAR                  H2CHRSB[11];
SQLCHAR                  H2CHRBIT[11];
SQLCHAR                  H2DDATE[11];
SQLCHAR                  H2TTIME[9];
SQLCHAR                  H2TSTMP[27];

SQLSMALLINT              I2SMINT;
SQLSMALLINT              I2INT4;
SQLSMALLINT              I2FLOAT8;
SQLSMALLINT              I2CHR10;
SQLSMALLINT              I2VCHR20;
SQLSMALLINT              I2LVCHR;
SQLSMALLINT              I2CHRSB;
SQLSMALLINT              I2CHRBIT;
SQLSMALLINT              I2DDATE;
SQLSMALLINT              I2TTIME;
SQLSMALLINT              I2TSTMP;

SQLINTEGER               LEN_H2SMINT;
SQLINTEGER               LEN_H2INT4;
SQLINTEGER               LEN_H2FLOAT8;
SQLINTEGER               LEN_H2CHR10;
SQLINTEGER               LEN_H2VCHR20;
SQLINTEGER               LEN_H2LVCHR;
SQLINTEGER               LEN_H2CHRSB;
SQLINTEGER               LEN_H2CHRBIT;
SQLINTEGER               LEN_H2DDATE;
SQLINTEGER               LEN_H2TTIME;
SQLINTEGER               LEN_H2TSTMP;
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 3 of 20)

```
   SQLCHAR locsite[18] =  "stlec1";
   SQLCHAR remsite[18] =  "stlec1b";

   SQLCHAR    spname[8];
   SQLINTEGER    ix,jx,locix;
   SQLINTEGER     result;
   SQLCHAR    state_blank[6] ="     ";

SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc);
SQLRETURN
prt_sqlca();
   /*******************************************************************/
   /* Main Program                                                    */
   /*******************************************************************/
SQLINTEGER
main(SQLINTEGER argc, SQLCHAR *argv[] )
{
 printf("\nSPD29 INITIALIZATION");
 scale = 0;
 rc=0;

 rc=0;
 SPCODE=0;

 /* argv0 = sp module name */
 if (argc != 2)
  {
   printf("SPD29 parm number error\n   ");
   printf("SPD29 EXPECTED =%d\n",3);
   printf("SPD29 received =%d\n",argc);
   goto dberror;
  }
  strcpy((char *)spname,(char *)argv[0]);
  result = strncmp((char *)spname,"SPD29",5);
  if (result != 0)
   {
    printf("SPD29 argv0 sp name  error\n   ");
    printf("SPD29 compare rusult =%i\n",result);
    printf("SPD29 expected =%s\n","SPD29");
    printf("SPD29 received spname=%s\n",spname);
    printf("SPD29 received argv0 =%s\n",argv[0]);
    goto dberror;
   }
  /* get input spcode value */
  SPCODE        = *(SQLINTEGER *)  argv[1];
 printf("\nSPD29 SQLAllocEnv        number=     1\n");
 henv=0;
 rc = SQLAllocEnv(&henv);
 if( rc != SQL_SUCCESS ) goto dberror;
 printf("\nSPD29-henv=%i",henv);
 /*****************************************************************/
 printf("\nSPD29 SQLAllocConnect                             ");
 hdbc=0;
 SQLAllocConnect(henv, &hdbc);
 if( rc != SQL_SUCCESS ) goto dberror;
 printf("\nSPD29-hdbc=%i",hdbc);
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 4 of 20)

```
/****************************************************************/
/* Make sure no autocommits after cursors are allocated, commits */
/* cause sp failure. AUTOCOMMIT=0 could also be specified in the */
/* INI file.                                                     */
/* Also, sp could be defined with COMMIT_ON_RETURN in the        */
/* DB2 catalog table SYSIBM.SYSPROCEDURES, but be wary that this */
/* removes control from the client appl to control commit scope. */
/****************************************************************/
printf("\nSPD29 SQLSetConnectOption-no autocommits in stored procs");
rc = SQLSetConnectOption(hdbc,SQL_AUTOCOMMIT,SQL_AUTOCOMMIT_OFF);
if( rc != SQL_SUCCESS ) goto dberror;
/****************************************************************/
printf("\nSPD29 SQLConnect  NULL connect in stored proc      ");
strcpy((char *)uid,"cliuser");
strcpy((char *)pwd,"password");
printf("\nSPD29 server=%s",NULL);
printf("\nSPD29 uid=%s",uid);
printf("\nSPD29 pwd=%s",pwd);
rc=SQLConnect(hdbc, NULL, 0, uid, SQL_NTS, pwd, SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;
/****************************************************************/
/* Start SQL statements *****************************************/
/****************************************************************/
switch(SPCODE)
{
 /***************************************************************/
 /* CASE(SPCODE=0) do nothing and return                   *****/
 /***************************************************************/
  case 0:
    break;
  case 1:
 /***************************************************************/
 /* CASE(SPCODE=1)                                         *****/
 /*  -sqlprepare/sqlexecute insert int4=200                *****/
 /*  -sqlexecdirect        insert int4=201                 *****/
 /* *validated in client appl that inserts occur           *****/
 /***************************************************************/
    SPCODE=0;

    printf("\nSPD29 SQLAllocStmt                               \n");
    hstmt=0;
    rc=SQLAllocStmt(hdbc, &hstmt);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nSPD29-hstmt=%i\n",hstmt);

    printf("\nSPD29 SQLPrepare                                 \n");
    strcpy((char *)sqlstmt,
    "insert into TABLE2A(int4) values(?)");
    printf("\nSPD29 sqlstmt=%s",sqlstmt);
    rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nSPD29 SQLNumParams                               \n");
    rc=SQLNumParams(hstmt,&pcpar);
    if( rc != SQL_SUCCESS) goto dberror;
    if (pcpar!=1) {
       printf("\nSPD29 incorrect pcpar=%d",pcpar);
       goto dberror;
      }
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 5 of 20)

```
   printf("\nSPD29 SQLBindParameter   int4                       \n");
   H1INT4=200;
   LEN_H1INT4=sizeof(H1INT4);
   rc=SQLBindParameter(hstmt,1,SQL_PARAM_INPUT,SQL_C_LONG,
   SQL_INTEGER,0,0,&H1INT4,0,(SQLINTEGER *)&LEN_H1INT4);
   if( rc != SQL_SUCCESS) goto dberror;

   printf("\nSPD29 SQLExecute                                   \n");
   rc=SQLExecute(hstmt);
   if( rc != SQL_SUCCESS) goto dberror;

   printf("\nSPD29  SQLFreeStmt                                  \n");
   rc=SQLFreeStmt(hstmt, SQL_DROP);
   if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
  printf("\nAPDIN SQLAllocStmt   stmt=%d",__LINE__);
  hstmt=0;
  rc=SQLAllocStmt(hdbc, &hstmt);
  if( rc != SQL_SUCCESS ) goto dberror;
  printf("\nAPDIN-hstmt=%i\n",hstmt);

  jx=201;
  printf("\nAPDIN SQLExecDirect  stmt=%d",__LINE__);
  strcpy((char *)sqlstmt,"insert into table2a values(");
  sprintf((char *)sqlstmt+strlen((char *)sqlstmt),"%ld",jx);
  strcat((char *)sqlstmt,
  ",4,8.2E+30,1515151515151.51,'CHAR','VCHAR','LVCCHAR','SBCS'");
    strcat((char *)sqlstmt,
  ",'MIXED','01/01/1991','3:33 PM','1999-09-09-09.09.09.090909')");
  printf("\nAPDIN sqlstmt=%s",sqlstmt);
  rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
  if( rc != SQL_SUCCESS ) goto dberror;

  break;
/*****************************************************************/
 case 2:
/****************************************************************/
/* CASE(SPCODE=2)                                          *****/
/*  -sqlprepare/sqlexecute select int4  from table2a       *****/
/*  -sqlprepare/sqlexecute select chr10 from table2a       *****/
/* *qrs cursors should be allocated and left open by CLI    *****/
/****************************************************************/
   SPCODE=0;

/* generate 1st query result set  */
  printf("\nSPD29 SQLAllocStmt                                  \n");
  hstmt=0;
  rc=SQLAllocStmt(hdbc, &hstmt);
  if( rc != SQL_SUCCESS ) goto dberror;
  printf("\nSPD29-hstmt=%i\n",hstmt);

  printf("\nSPD29 SQLPrepare                                    \n");
  strcpy((char *)sqlstmt,
   "SELECT INT4 FROM TABLE2A");
  printf("\nSPD29 sqlstmt=%s",sqlstmt);
  rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
  if( rc != SQL_SUCCESS ) goto dberror;

  printf("\nSPD29 SQLExeccute                                   \n");
  rc=SQLExecute(hstmt);
  if( rc != SQL_SUCCESS ) goto dberror;
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 6 of 20)

```
 /* allocate 2nd stmt handle for 2nd queryresultset */
 /* generate 2nd  queryresultset  */
   printf("\nSPD29 SQLAllocStmt                                \n");
   hstmt=0;
   rc=SQLAllocStmt(hdbc, &hstmt2);
   if( rc != SQL_SUCCESS ) goto dberror;
   printf("\nSPD29-hstmt2=%i\n",hstmt2);

   printf("\nSPD29 SQLPrepare                                  \n");
   strcpy((char *)sqlstmt2,
    "SELECT CHR10 FROM TABLE2A");
   printf("\nSPD29 sqlstmt2=%s",sqlstmt2);
   rc=SQLPrepare(hstmt2,sqlstmt2,SQL_NTS);
   if( rc != SQL_SUCCESS ) goto dberror;

   printf("\nSPD29 SQLExeccute                                 \n");
   rc=SQLExecute(hstmt2);
   if( rc != SQL_SUCCESS ) goto dberror;

   /*leave queryresultset cursor open for fetch back at client appl */

   break;
 /*****************************************************************/
  default:
   {
    printf("SPD29 INPUT SPCODE INVALID\n");
    printf("SPD29...EXPECTED SPCODE=0-2\n");
    printf("SPD29...RECEIVED SPCODE=%i\n",SPCODE);
    goto dberror;
    break;
    }
}
/******************************************************************/
/* End SQL statements ********************************************/
/******************************************************************/
/*Be sure NOT to put a SQLTransact with SQL_COMMIT in a DB2 or   */
/*  z/OS stored procedure.  Commit is not allowed in a DB2 or    */
/*  z/OS stored procedure.  Use SQLTransact with SQL_ROLLBACK to */
/*  force a must rollback condition for this sp and calling      */
/*  client application.                                          */
/******************************************************************/
printf("\nSPD29 SQLDisconnect      number=    4\n");
rc=SQLDisconnect(hdbc);
if( rc != SQL_SUCCESS ) goto dberror;
/******************************************************************/
printf("\nSPD29 SQLFreeConnect     number=    5\n");
rc = SQLFreeConnect(hdbc);
if( rc != SQL_SUCCESS ) goto dberror;
/******************************************************************/
printf("\nSPD29 SQLFreeEnv         number=    6\n");
rc = SQLFreeEnv(henv);
if( rc != SQL_SUCCESS ) goto dberror;
/******************************************************************/
goto pgmend;

dberror:
printf("\nSPD29 entry dberror label");
printf("\nSPD29 rc=%d",rc);
check_error(henv,hdbc,hstmt,rc);
printf("\nSPD29 SQLFreeEnv         number=    7\n");
rc = SQLFreeEnv(henv);
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 7 of 20)

```
 printf("\nSPD29 rc=%d",rc);
 rc=12;
 rc=12;
 SPCODE=12;
 goto pgmend;

 pgmend:

 printf("\nSPD29  TERMINATION   ");
 if (rc!=0)
  {
  printf("\nSPD29 WAS NOT SUCCESSFUL");
  printf("\nSPD29 SPCODE  = %i", SPCODE      );
  printf("\nSPD29 rc          = %i", rc  );
  }
 else
 {
  printf("\nSPD29 WAS SUCCESSFUL");
 }
 /* assign output spcode value */
 *(SQLINTEGER  *) argv[1] = SPCODE;
 exit;
}  /*END MAIN*/
/*******************************************************************
** check_error - call print_error(), checks severity of return code
*******************************************************************/
SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc )
{
    SQLCHAR         buffer[SQL_MAX_MESSAGE_LENGTH + 1];
    SQLCHAR         cli_sqlstate[SQL_SQLSTATE_SIZE + 1];
    SQLINTEGER     cli_sqlcode;
    SQLSMALLINT   length;

    printf("\nSPD29 entry check_error rtn");

    switch (frc) {
    case SQL_SUCCESS:
        break;
    case SQL_INVALID_HANDLE:
        printf("\nSPD29 check_error> SQL_INVALID HANDLE ");
    case SQL_ERROR:
        printf("\nSPD29 check_error> SQL_ERROR ");
        break;
    case SQL_SUCCESS_WITH_INFO:
        printf("\nSPD29 check_error>  SQL_SUCCESS_WITH_INFO");
        break;
    case SQL_NO_DATA_FOUND:
        printf("\nSPD29 check_error> SQL_NO_DATA_FOUND ");
        break;
    default:
        printf("\nSPD29 check_error> Invalid rc from api rc=%i",frc);
        break;
    } /*end switch*/
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 8 of 20)

```
    printf("\nSPD29 SQLError  ");
    while ((rc=SQLError(henv, hdbc, hstmt, cli_sqlstate, &cli_sqlcode,
      buffer,SQL_MAX_MESSAGE_LENGTH + 1, &length)) == SQL_SUCCESS) {
        printf("        SQLSTATE: %s", cli_sqlstate);
        printf("Native Error Code: %ld", cli_sqlcode);
        printf("%s ", buffer);
    };
    if (rc!=SQL_NO_DATA_FOUND)
      printf("SQLError api call failed rc=%d",rc);

    printf("\nSPD29 SQLGetSQLCA  ");
    rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    if( rc == SQL_SUCCESS )
      prt_sqlca();
    else
      printf("\n  SPD29-check_error SQLGetSQLCA failed rc=%i",rc);

    return (frc);
}
  /******************************************************************/
  /*                P r i n t   S Q L C A                           */
  /******************************************************************/
SQLRETURN
  prt_sqlca()
  {
    SQLINTEGER i;
    printf("\nlSPD29 entry prts_sqlca rtn");
    printf("\r\r*** Printing the SQLCA:\r");
    printf("\nSQLCAID .... %s",sqlca.sqlcaid);
    printf("\nSQLCABC .... %d",sqlca.sqlcabc);
    printf("\nSQLCODE .... %d",sqlca.sqlcode);
    printf("\nSQLERRML ... %d",sqlca.sqlerrml);
    printf("\nSQLERRMC ... %s",sqlca.sqlerrmc);
    printf("\nSQLERRP  ... %s",sqlca.sqlerrp);
    for (i = 0; i < 6; i++)
      printf("\nSQLERRD%d ... %d",i+1,sqlca.sqlerrd??(i??));
    for (i = 0; i < 10; i++)
      printf("\nSQLWARN%d ... %c",i,sqlca.sqlwarn[i]);
    printf("\nSQLWARNA ... %c",sqlca.sqlwarn[10]);
    printf("\nSQLSTATE ... %s",sqlca.sqlstate);

    return(0);
  }                                             /* End of prtsqlca */
  /******************************************************************/
  /*END OF SPD29 ****************************************************/
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 9 of 20)

STEP 5. Client application

```
 /******************************************************************/
 /*START OF SPD29**************************************************/
 /*  SCEANRIO PSEUDOCODE:                                          */
 /*  APD29(CLI CODE CLIENT APPL)                                   */
 /*       -CALL SPD29 (CLI CODE STORED PROCEDURE APPL)              */
 /*         -SPCODE=0                                              */
 /*             -PRINTF MSGS (CHECK SDSF FOR SPAS ADDR TO VERFIFY)   */
 /*         -SPCODE=1                                              */
 /*             -PRINTF MSGS (CHECK SDSF FOR SPAS ADDR TO VERFIFY)   */
 /*             -SQLPREPARE/EXECUTE INSERT INT4=200                */
 /*             -SQLEXECDIRECT      INSERT INT4=201                */
 /*         -SPCODE=2                                              */
 /*             -PRINTF MSGS (CHECK SDSF FOR SPAS ADDR TO VERFIFY)   */
 /*             -SQLPREPARE/EXECUTE SELECT INT4 FROM TABLE2A         */
 /*             -SQLPREPARE/EXECUTE SELECT CHR10 FROM TABLE2A        */
 /*              (CLI CURSORS OPENED 'WITH RETURN')...             */
 /*         -RETURN                                                */
 /*       -FETCH QRS FROM SP CURSOR                                */
 /*       -COMMIT                                                  */
 /*       -VERFIFY INSERTS BY SPD29                                */
 /******************************************************************/
   /* Include the 'C' include files                                */
   /****************************************************************/
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
   #include "sqlcli1.h"
   #include <sqlca.h>
   /****************************************************************/
   /* Variables for COMPARE routines                               */
   /****************************************************************/
#ifndef NULL
#define NULL   0
#endif

    SQLHENV henv = SQL_NULL_HENV;
    SQLHDBC hdbc = SQL_NULL_HDBC;
    SQLHSTMT hstmt = SQL_NULL_HSTMT;
    SQLRETURN rc = SQL_SUCCESS;
    SQLINTEGER      id;
    SQLCHAR         name[51];
    SQLINTEGER      namelen, intlen, colcount;
    SQLSMALLINT     scale;
    struct sqlca    sqlca;
    SQLCHAR   server[18];
    SQLCHAR   uid[30];
    SQLCHAR   pwd[30];
    SQLCHAR   sqlstmt[250];
    SQLSMALLINT  pcpar=0;
    SQLSMALLINT  pccol=0;

   SQLINTEGER    SPCODE;
   struct {
     SQLSMALLINT LEN;
     SQLCHAR  DATA[200]; }         STMTSQL;
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 10 of 20)

```
SQLSMALLINT                H1SMINT;
SQLINTEGER                 H1INT4;
SQLDOUBLE              H1FLOAT8;
SQLDOUBLE              H1DEC312;
SQLCHAR                 H1CHR10[11];
SQLCHAR                 H1VCHR20[21];
SQLCHAR                 H1LVCHR[21];
SQLCHAR                 H1CHRSB[11];
SQLCHAR                 H1CHRBIT[11];
SQLCHAR                 H1DDATE[11];
SQLCHAR                 H1TTIME[9];
SQLCHAR                 H1TSTMP[27];

SQLSMALLINT                I1SMINT;
SQLSMALLINT                I1INT4;
SQLSMALLINT                I1FLOAT8;
SQLSMALLINT                I1DEC312;
SQLSMALLINT                I1CHR10;
SQLSMALLINT                I1VCHR20;
SQLSMALLINT                I1LVCHR;
SQLSMALLINT                I1CHRSB;
SQLSMALLINT                I1CHRBIT;
SQLSMALLINT                I1DDATE;
SQLSMALLINT                I1TTIME;
SQLSMALLINT                I1TSTMP;

SQLINTEGER                 LNH1SMINT;
SQLINTEGER                 LNH1INT4;
SQLINTEGER                 LNH1FLOAT8;
SQLINTEGER                 LNH1DEC312;
SQLINTEGER                 LNH1CHR10;
SQLINTEGER                 LNH1VCHR20;
SQLINTEGER                 LNH1LVCHR;
SQLINTEGER                 LNH1CHRSB;
SQLINTEGER                 LNH1CHRBIT;
SQLINTEGER                 LNH1DDATE;
SQLINTEGER                 LNH1TTIME;
SQLINTEGER                 LNH1TSTMP;

SQLSMALLINT                H2SMINT;
SQLINTEGER                 H2INT4;
SQLDOUBLE               H2FLOAT8;
SQLCHAR                 H2CHR10[11];
SQLCHAR                 H2VCHR20[21];
SQLCHAR                 H2LVCHR[21];
SQLCHAR                 H2CHRSB[11];
SQLCHAR                 H2CHRBIT[11];
SQLCHAR                 H2DDATE[11];
SQLCHAR                 H2TTIME[9];
SQLCHAR                 H2TSTMP[27];

SQLSMALLINT                I2SMINT;
SQLSMALLINT                I2INT4;
SQLSMALLINT                I2FLOAT8;
SQLSMALLINT                I2CHR10;
SQLSMALLINT                I2VCHR20;
SQLSMALLINT                I2LVCHR;
SQLSMALLINT                I2CHRSB;
SQLSMALLINT                I2CHRBIT;
SQLSMALLINT                I2DDATE;
SQLSMALLINT                I2TTIME;
SQLSMALLINT                I2TSTMP;
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 11 of 20)

```
   SQLINTEGER                  LNH2SMINT;
   SQLINTEGER                  LNH2INT4;
   SQLINTEGER                  LNH2FLOAT8;
   SQLINTEGER                  LNH2CHR10;
   SQLINTEGER                  LNH2VCHR20;
   SQLINTEGER                  LNH2LVCHR;
   SQLINTEGER                  LNH2CHRSB;
   SQLINTEGER                  LNH2CHRBIT;
   SQLINTEGER                  LNH2DDATE;
   SQLINTEGER                  LNH2TTIME;
   SQLINTEGER                  LNH2TSTMP;

   SQLCHAR locsite[18] =  "stlec1";
   SQLCHAR remsite[18] =  "stlec1b";

   SQLINTEGER    ix,jx,locix;
   SQLINTEGER     result;
   SQLCHAR    state_blank[6] ="     ";

SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc);
SQLRETURN
prt_sqlca();
   /*****************************************************************/
   /* Main Program                                                  */
   /*****************************************************************/
SQLINTEGER
main()
{
    printf("\nAPD29 INITIALIZATION");
    scale = 0;
    rc=0;

    printf("\nAPD29 SQLAllocEnv   stmt=%d",__LINE__);
    henv=0;
    rc = SQLAllocEnv(&henv);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPD29-henv=%i",henv);

for (locix=1;locix<=2;locix++)
{
 /* Start SQL statements ****************************************/
 /**************************************************************/
    printf("\nAPD29 SQLAllocConnect                            ");
    hdbc=0;
    SQLAllocConnect(henv, &hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPD29-hdbc=%i",hdbc);
 /**************************************************************/
    printf("\nAPD29 SQLConnect                                 ");
    if (locix == 1)
    {
     strcpy((char *)server,(char *)locsite);
    }
    else
     {
      strcpy((char *)server,(char *)remsite);
     }
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 12 of 20)

```
   strcpy((char *)uid,"cliuser");
   strcpy((char *)pwd,"password");
   printf("\nAPD29 server=%s",server);
   printf("\nAPD29 uid=%s",uid);
   printf("\nAPD29 pwd=%s",pwd);
   rc=SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
   if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
/* CASE(SPCODE=0)  QRS RETURNED=0  COL=0  ROW=0                  */
/*****************************************************************/
printf("\nAPD29 SQLAllocStmt   stmt=%d",__LINE__);
hstmt=0;
rc=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
printf("\nAPD29-hstmt=%i\n",hstmt);

SPCODE=0;
printf("\nAPD29 call sp SPCODE =%i\n",SPCODE);
printf("\nAPD29 SQLPrepare  stmt=%d",__LINE__);
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=%s",sqlstmt);
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLBindParameter  stmt=%d",__LINE__);
rc = SQLBindParameter(hstmt,
                      1,
                      SQL_PARAM_INPUT_OUTPUT,
                      SQL_C_LONG,
                      SQL_INTEGER,
                      0,
                      0,
                      &SPCODE,
                      0,
                      NULL);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLExecute  stmt=%d",__LINE__);
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
if( SPCODE != 0 )
{
  printf("\nAPD29 SPCODE not zero, spcode=%i\n",SPCODE);
  goto dberror;
}

printf("\nAPD29 SQLTransact stmt=%d",__LINE__);
rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29  SQLFreeStmt stmt=%d",__LINE__);
rc=SQLFreeStmt(hstmt, SQL_DROP);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
/* CASE(SPCODE=1)  QRS RETURNED=0  COL=0  ROW=0                  */
/*****************************************************************/
printf("\nAPD29 SQLAllocStmt   stmt=%d",__LINE__);
hstmt=0;
rc=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
printf("\nAPD29-hstmt=%i\n",hstmt);
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 13 of 20)

```
SPCODE=1;
printf("\nAPD29 call sp SPCODE =%i\n",SPCODE);
printf("\nAPD29 SQLPrepare  stmt=%d",__LINE__);
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=%s",sqlstmt);
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLBindParameter  stmt=%d",__LINE__);
rc = SQLBindParameter(hstmt,
                      1,
                      SQL_PARAM_INPUT_OUTPUT,
                      SQL_C_LONG,
                      SQL_INTEGER,
                      0,
                      0,
                      &SPCODE,
                      0,
                      NULL);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLExecute  stmt=%d",__LINE__);
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
if( SPCODE != 0 )
{
  printf("\nAPD29 SPCODE not zero, spcode=%i\n",SPCODE);
  goto dberror;
}

printf("\nAPD29 SQLTransact stmt=%d",__LINE__);
rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29  SQLFreeStmt stmt=%d",__LINE__);
rc=SQLFreeStmt(hstmt, SQL_DROP);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
/* CASE(SPCODE=2)  QRS RETURNED=2  COL=1(int4/chr10)  ROW=100+   */
/*****************************************************************/
printf("\nAPD29 SQLAllocStmt         number=   18\n");
hstmt=0;
rc=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
printf("\nAPD29-hstmt=%i\n",hstmt);

SPCODE=2;
printf("\nAPD29 call sp SPCODE =%i\n",SPCODE);
printf("\nAPD29 SQLPrepare           number=   19\n");
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=%s",sqlstmt);
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 14 of 20)

```
printf("\nAPD29 SQLBindParameter     number=   20\n");
rc = SQLBindParameter(hstmt,
                      1,
                      SQL_PARAM_INPUT_OUTPUT,
                      SQL_C_LONG,
                      SQL_INTEGER,
                      0,
                      0,
                      &SPCODE,
                      0,
                      NULL);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLExecute           number=   21\n");
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
if( SPCODE != 0 )
{
  printf("\nAPD29 spcode incorrect");
  goto dberror;
}

 printf("\nAPD29 SQLNumResultCols     number=   22\n");
 rc=SQLNumResultCols(hstmt,&pccol);
 if (pccol!=1)
 {
   printf("APD29 col count wrong=%i\n",pccol);
   goto dberror;
 }

printf("\nAPD29 SQLBindCol           number=   23\n");
rc=SQLBindCol(hstmt,
              1,
              SQL_C_LONG,
              (SQLPOINTER) &H1INT4,
              (SQLINTEGER)sizeof(SQLINTEGER),
              (SQLINTEGER *) &LNH1INT4       );
if( rc != SQL_SUCCESS ) goto dberror;

jx=0;
printf("\nAPD29 SQLFetch             number=   24\n");
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
{
 jx++;
 printf("\nAPD29 fetch loop jx =%i\n",jx);
 if ( (H1INT4<=0) ||  (H1INT4>=202)
      || (LNH1INT4!=4 && LNH1INT4!=-1)  )
  { /* data error */
    printf("\nAPD29 H1INT4=%i\n",H1INT4);
    printf("\nAPD29 LNH1INT4=%i\n",LNH1INT4);
    goto dberror;
  }
 printf("\nAPD29 SQLFetch             number=   24\n");
} /* end while loop */

if( rc != SQL_NO_DATA_FOUND )
{
  printf("\nAPD29 invalid end of data\n");
  goto dberror;
}
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 15 of 20)

```
printf("\nAPD29 SQLMoreResults       number=   25\n");
rc=SQLMoreResults(hstmt);
if(rc != SQL_SUCCESS) goto dberror;

printf("\nAPD29 SQLNumResultCols     number=   26\n");
rc=SQLNumResultCols(hstmt,&pccol);
if (pccol!=1) {
  printf("APD29 col count wrong=%i\n",pccol);
  goto dberror;
}

printf("\nAPD29 SQLBindCol           number=   27\n");
rc=SQLBindCol(hstmt,
              1,
              SQL_C_CHAR,
              (SQLPOINTER) H1CHR10,
              (SQLINTEGER)sizeof(H1CHR10),
              (SQLINTEGER *) &LNH1CHR10       );
if( rc != SQL_SUCCESS ) goto dberror;

jx=0;
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
{
 jx++;
 printf("\nAPD29 fetch loop jx =%i\n",jx);
  result=strcmp((char *)H1CHR10,"CHAR      ");
  if ( (result!=0)
      || (LNH1INT4!=4 && LNH1INT4!=-1)  )
   {
    printf("\nAPD29 H1CHR10=%s\n",H1CHR10);
    printf("\nAPD29 result=%i\n",result);
    printf("\nAPD29 LNH1CHR10=%i\n",LNH1CHR10);
    printf("\nAPD29 strlen(H1CHR10)=%i\n",strlen((char *)H1CHR10));
    goto dberror;
   }
 printf("\nAPD29 SQLFetch             number=   24\n");
} /* end while loop */

if( rc != SQL_NO_DATA_FOUND )
  goto dberror;

printf("\nAPD29 SQLMoreResults       number=   29\n");
rc=SQLMoreResults(hstmt);
if( rc != SQL_NO_DATA_FOUND) goto dberror;

printf("\nAPD29 SQLTransact          number=   30\n");
rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29  SQLFreeStmt         number=   31\n");
rc=SQLFreeStmt(hstmt, SQL_DROP);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
   printf("\nAPD29 SQLDisconnect  stmt=%d",__LINE__);
   rc=SQLDisconnect(hdbc);
   if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
   printf("\nSQLFreeConnect  stmt=%d",__LINE__);
   rc=SQLFreeConnect(hdbc);
   if( rc != SQL_SUCCESS ) goto dberror;
 /*****************************************************************/
 /* End SQL statements *******************************************/
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 16 of 20)

```
} /* end for each site perform these stmts */

for (locix=1;locix<=2;locix++)
{
 /*****************************************************************/
    printf("\nAPD29 SQLAllocConnect                               ");
    hdbc=0;
    SQLAllocConnect(henv, &hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPD29-hdbc=%i",hdbc);
 /*****************************************************************/
    printf("\nAPD29 SQLConnect                                    ");
    if (locix == 1)
    {
     strcpy((char *)server,(char *)locsite);
    }
    else
     {
      strcpy((char *)server,(char *)remsite);
     }

    strcpy((char *)uid,"cliuser");
    strcpy((char *)pwd,"password");
    printf("\nAPD29 server=%s",server);
    printf("\nAPD29 uid=%s",uid);
    printf("\nAPD29 pwd=%s",pwd);
    rc=SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;
 /*****************************************************************/
 /* Start validate SQL statements *********************************/
 /*****************************************************************/
    printf("\nAPD01 SQLAllocStmt                                   \n");
    hstmt=0;
    rc=SQLAllocStmt(hdbc, &hstmt);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPD01-hstmt=%i\n",hstmt);

    printf("\nAPD01 SQLExecDirect                                  \n");
    strcpy((char *)sqlstmt,
    "SELECT INT4 FROM TABLE2A WHERE INT4=200");
    printf("\nAPD01 sqlstmt=%s",sqlstmt);
    rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nAPD01 SQLBindCol                                     \n");
    rc=SQLBindCol(hstmt,
                  1,
                  SQL_C_LONG,
                  (SQLPOINTER) &H1INT4,;
                  (SQLINTEGER)sizeof(SQLINTEGER),
                  (SQLINTEGER *) &LNH1INT4       );
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nAPD01 SQLFetch                                       \n");
    rc=SQLFetch(hstmt);
    if( rc != SQL_SUCCESS ) goto dberror;
    if ((H1INT4!=200) || (LNH1INT4!=4))
     {
     printf("\nAPD01 H1INT4=%i\n",H1INT4);
     printf("\nAPD01 LNH1INT4=%i\n",LNH1INT4);
     goto dberror;
     }
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 17 of 20)

```
   printf("\nAPD01 SQLTransact                                  \n");
   rc=SQLTransact(henv, hdbc, SQL_COMMIT);
   if( rc != SQL_SUCCESS ) goto dberror;

   printf("\nAPD01  SQLFreeStmt                                 \n");
   rc=SQLFreeStmt(hstmt, SQL_CLOSE);
   if( rc != SQL_SUCCESS ) goto dberror;

   printf("\nAPD01 SQLExecDirect                                \n");
   strcpy((char *)sqlstmt,
   "SELECT INT4 FROM TABLE2A WHERE INT4=201");
   printf("\nAPD01 sqlstmt=%s",sqlstmt);
   rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
   if( rc != SQL_SUCCESS ) goto dberror;

   printf("\nAPD01 SQLFetch                                     \n");
   rc=SQLFetch(hstmt);
   if( rc != SQL_SUCCESS ) goto dberror;
   if ((H1INT4!=201) || (LNH1INT4!=4))
    {
    printf("\nAPD01 H1INT4=%i\n",H1INT4);
    printf("\nAPD01 LNH1INT4=%i\n",LNH1INT4);
    goto dberror;
    }

   printf("\nAPD01 SQLTransact                                  \n");
   rc=SQLTransact(henv, hdbc, SQL_COMMIT);
   if( rc != SQL_SUCCESS ) goto dberror;

   printf("\nAPD01  SQLFreeStmt                                 \n");
   rc=SQLFreeStmt(hstmt, SQL_DROP);
   if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
/* End validate SQL statements ***********************************/
/*****************************************************************/
   printf("\nAPD29 SQLDisconnect  stmt=%d",__LINE__);
   rc=SQLDisconnect(hdbc);
   if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
   printf("\nSQLFreeConnect  stmt=%d",__LINE__);
   rc=SQLFreeConnect(hdbc);
   if( rc != SQL_SUCCESS ) goto dberror;

} /* end for each site perform these stmts */
/*****************************************************************/
   printf("\nSQLFreeEnv  stmt=%d",__LINE__);
   rc=SQLFreeEnv(henv);
   if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
goto pgmend;

dberror:
printf("\nAPD29 entry dberror label");
printf("\nAPD29 rc=%d",rc);
check_error(henv,hdbc,hstmt,rc);
printf("\nAPDXX SQLFreeEnv         number=     6\n");
rc=SQLFreeEnv(henv);
printf("\nAPDXX FREEENV rc =%d",rc);
rc=12;
printf("\nAPDXX DBERROR set rc =%d",rc);
goto pgmend;
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 18 of 20)

```
 pgmend:
     printf("\nAPD29  TERMINATION   ");
     if (rc!=0)
      {
      printf("\nAPD29 WAS NOT SUCCESSFUL");
      printf("\nAPD29 SPCODE  = %i", SPCODE      );
      printf("\nAPD29 rc          = %i", rc  );
      }
     else
      printf("\nAPD29 WAS SUCCESSFUL");

     return(rc);

}  /*END MAIN*/
/*******************************************************************
** check_error - call print_error(), checks severity of return code
*******************************************************************/
SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc )
{

   SQLCHAR         buffer_SQL_MAX_MESSAGE_LENGTH + 1¨;
    SQLCHAR         cli_sqlstate_SQL_SQLSTATE_SIZE + 1¨;
    SQLINTEGER     cli_sqlcode;
    SQLSMALLINT   length;

    printf("\nAPD29 entry check_error rtn");

    switch (frc) {
    case SQL_SUCCESS:
        break;
    case SQL_INVALID_HANDLE:
        printf("\nAPD29 check_error> SQL_INVALID HANDLE ");
    case SQL_ERROR:
        printf("\nAPD29 check_error> SQL_ERROR ");
        break;
    case SQL_SUCCESS_WITH_INFO:
        printf("\nAPD29 check_error>  SQL_SUCCESS_WITH_INFO");
        break;
    case SQL_NO_DATA_FOUND:
        printf("\nAPD29 check_error> SQL_NO_DATA_FOUND ");
        break;
    default:
        printf("\nAPD29 check_error> Invalid rc from api rc=%i",frc);
        break;
    } /*end switch*/

    printf("\nAPD29 SQLError  ");
    while ((rc=SQLError(henv, hdbc, hstmt, cli_sqlstate, &cli_sqlcode,
      buffer,SQL_MAX_MESSAGE_LENGTH + 1, &length)) == SQL_SUCCESS) {
        printf("         SQLSTATE: %s", cli_sqlstate);
        printf("Native Error Code: %ld", cli_sqlcode);
        printf("%s ", buffer);
    };
    if (rc!=SQL_NO_DATA_FOUND)
      printf("SQLError api call failed rc=%d",rc);
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 19 of 20)

```
    printf("\nAPD29 SQLGetSQLCA  ");
    rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    if( rc == SQL_SUCCESS )
      prt_sqlca();
    else
      printf("\n  APD29-check_error SQLGetSQLCA failed rc=%i",rc);

    return (frc);
}
  /*******************************************************************/
  /*                 P r i n t   S Q L C A                          */
  /*******************************************************************/
SQLRETURN
  prt_sqlca()
  {
    SQLINTEGER i;
    printf("\nlAPD29 entry prts_sqlca rtn");
    printf("\r\r*** Printing the SQLCA:\r");
    printf("\nSQLCAID .... %s",sqlca.sqlcaid);
    printf("\nSQLCABC .... %d",sqlca.sqlcabc);
    printf("\nSQLCODE .... %d",sqlca.sqlcode);
    printf("\nSQLERRML ... %d",sqlca.sqlerrml);
    printf("\nSQLERRMC ... %s",sqlca.sqlerrmc);
    printf("\nSQLERRP  ... %s",sqlca.sqlerrp);
    for (i = 0; i < 6; i++)
      printf("\nSQLERRD%d ... %d",i+1,sqlca.sqlerrd??(i??));
    for (i = 0; i < 10; i++)
      printf("\nSQLWARN%d ... %c",i,sqlca.sqlwarn[i]);
    printf("\nSQLWARNA ... %c",sqlca.sqlwarn[10]);
    printf("\nSQLSTATE ... %s",sqlca.sqlstate);

    return(0);
  }                                                /* End of prtsqlca */
  /*END OF APD29*****************************************************/
```

Figure 64. A client application that calls a DB2 ODBC stored procedure (Part 20 of 20)

Notices

Programming interface information

This book is intended to help you to write applications that use DB2 ODBC to access IBM DB2 UDB for z/OS servers. This book documents General-use Programming Interface and Associated Guidance Information provided by DB2 UDB for z/OS.

General-use programming interfaces allow you to write programs that obtain the services of DB2 UDB for z/OS.

Trademarks

Glossary

The following terms and abbreviations are defined as they are used in the DB2 library.

A

abend. Abnormal end of task.

abend reason code. A 4-byte hexadecimal code that uniquely identifies a problem with DB2. A complete list of DB2 abend reason codes and their explanations is contained in *DB2 Messages and Codes*.

abnormal end of task (abend). Termination of a task, job, or subsystem because of an error condition that recovery facilities cannot resolve during execution.

access method services. The facility that is used to define and reproduce VSAM key-sequenced data sets.

access path. The path that is used to locate data that is specified in SQL statements. An access path can be indexed or sequential.

active log. The portion of the DB2 log to which log records are written as they are generated. The active log always contains the most recent log records, whereas the archive log holds those records that are older and no longer fit on the active log.

active member state. A state of a member of a data sharing group. The cross-system coupling facility identifies each active member with a group and associates the member with a particular task, address space, and z/OS system. A member that is not active has either a failed member state or a quiesced member state.

address space. A range of virtual storage pages that is identified by a number (ASID) and a collection of segment and page tables that map the virtual pages to real pages of the computer's memory.

address space connection. The result of connecting an allied address space to DB2. Each address space that contains a task that is connected to DB2 has exactly one address space connection, even though more than one task control block (TCB) can be present. See also *allied address space* and *task control block*.

| **address space identifier (ASID).** A unique
| system-assigned identifier for and address space.

administrative authority. A set of related privileges that DB2 defines. When you grant one of the administrative authorities to a person's ID, the person has all of the privileges that are associated with that administrative authority.

after trigger. A trigger that is defined with the trigger activation time AFTER.

agent. As used in DB2, the structure that associates all processes that are involved in a DB2 unit of work. An *allied agent* is generally synonymous with an *allied thread*. *System agents* are units of work that process tasks that are independent of the allied agent, such as prefetch processing, deferred writes, and service tasks.

alias. An alternative name that can be used in SQL statements to refer to a table or view in the same or a remote DB2 subsystem.

allied address space. An area of storage that is external to DB2 and that is connected to DB2. An allied address space is capable of requesting DB2 services.

allied thread. A thread that originates at the local DB2 subsystem and that can access data at a remote DB2 subsystem.

allocated cursor. A cursor that is defined for stored procedure result sets by using the SQL ALLOCATE CURSOR statement.

already verified. An LU 6.2 security option that allows DB2 to provide the user's verified authorization ID when allocating a conversation. With this option, the user is not validated by the partner DB2 subsystem.

ambiguous cursor. A database cursor that is in a plan or package that contains either PREPARE or EXECUTE IMMEDIATE SQL statements, and for which the following statements are true: the cursor is not defined with the FOR READ ONLY clause or the FOR UPDATE OF clause; the cursor is not defined on a read-only result table; the cursor is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

ANSI. American National Standards Institute.

APAR. Authorized program analysis report.

APAR fix corrective service. A temporary correction of an IBM software defect. The correction is temporary, because it is usually replaced at a later date by a more permanent correction, such as a program temporary fix (PTF).

APF. Authorized program facility.

API. Application programming interface.

APPL. A VTAM® network definition statement that is used to define DB2 to VTAM as an application program that uses SNA LU 6.2 protocols.

application. A program or set of programs that performs a task; for example, a payroll application.

application-directed connection. A connection that an application manages using the SQL CONNECT statement.

application plan. The control structure that is produced during the bind process. DB2 uses the application plan to process SQL statements that it encounters during statement execution.

application process. The unit to which resources and locks are allocated. An application process involves the execution of one or more programs.

application programming interface (API). A functional interface that is supplied by the operating system or by a separately orderable licensed program that allows an application program that is written in a high-level language to use specific data or functions of the operating system or licensed program.

application requester. The component on a remote system that generates DRDA requests for data on behalf of an application. An application requester accesses a DB2 database server using the DRDA application-directed protocol.

application server. The target of a request from a remote application. In the DB2 environment, the application server function is provided by the distributed data facility and is used to access DB2 data from remote applications.

archive log. The portion of the DB2 log that contains log records that have been copied from the active log.

ASCII. An encoding scheme that is used to represent strings in many environments, typically on PCs and workstations. Contrast with *EBCDIC* and *Unicode*.

| **ASID.** Address space identifier.

attachment facility. An interface between DB2 and TSO, IMS, CICS, or batch address spaces. An attachment facility allows application programs to access DB2.

attribute. A characteristic of an entity. For example, in database design, the phone number of an employee is one of that employee's attributes.

authorization ID. A string that can be verified for connection to DB2 and to which a set of privileges is allowed. It can represent an individual, an organizational group, or a function, but DB2 does not determine this representation.

authorized program analysis report (APAR). A report of a problem that is caused by a suspected defect in a current release of an IBM supplied program.

authorized program facility (APF). A facility that permits the identification of programs that are authorized to use restricted functions.

automatic query rewrite. A process that examines an SQL statement that refers to one or more base tables, and, if appropriate, rewrites the query so that it performs better. This process can also determine whether to rewrite a query so that it refers to one or more materialized query tables that are derived from the source tables.

auxiliary index. An index on an auxiliary table in which each index entry refers to a LOB.

auxiliary table. A table that stores columns outside the table in which they are defined. Contrast with *base table*.

B

backout. The process of undoing uncommitted changes that an application process made. This might be necessary in the event of a failure on the part of an application process, or as a result of a deadlock situation.

backward log recovery. The fourth and final phase of restart processing during which DB2 scans the log in a backward direction to apply UNDO log records for all aborted changes.

base table. (1) A table that is created by the SQL CREATE TABLE statement and that holds persistent data. Contrast with *result table* and *temporary table*.

(2) A table containing a LOB column definition. The actual LOB column data is not stored with the base table. The base table contains a row identifier for each row and an indicator column for each of its LOB columns. Contrast with *auxiliary table*.

base table space. A table space that contains base tables.

basic predicate. A predicate that compares two values.

basic sequential access method (BSAM). An access method for storing or retrieving data blocks in a continuous sequence, using either a sequential-access or a direct-access device.

| **batch message processing program.** In IMS, an
| application program that can perform batch-type
| processing online and can access the IMS input and
| output message queues.

before trigger. A trigger that is defined with the trigger activation time BEFORE.

binary integer. A basic data type that can be further classified as small integer or large integer.

binary large object (BLOB). A sequence of bytes where the size of the value ranges from 0 bytes to 2 GB−1. Such a string does not have an associated CCSID.

binary string. A sequence of bytes that is not associated with a CCSID. For example, the BLOB data type is a binary string.

bind. The process by which the output from the SQL precompiler is converted to a usable control structure, often called an access plan, application plan, or package. During this process, access paths to the data are selected and some authorization checking is performed. The types of bind are:

automatic bind. (More correctly, *automatic rebind*) A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.

dynamic bind. A process by which SQL statements are bound as they are entered.

incremental bind. A process by which SQL statements are bound during the execution of an application process.

static bind. A process by which SQL statements are bound after they have been precompiled. All static SQL statements are prepared for execution at the same time.

bit data. Data that is character type CHAR or VARCHAR and is not associated with a coded character set.

BLOB. Binary large object.

block fetch. A capability in which DB2 can retrieve, or fetch, a large set of rows together. Using block fetch can significantly reduce the number of messages that are being sent across the network. Block fetch applies only to cursors that do not update data.

BMP. Batch Message Processing (IMS). See *batch message processing program*.

bootstrap data set (BSDS). A VSAM data set that contains name and status information for DB2, as well as RBA range specifications, for all active and archive log data sets. It also contains passwords for the DB2 directory and catalog, and lists of conditional restart and checkpoint records.

BSAM. Basic sequential access method.

BSDS. Bootstrap data set.

buffer pool. Main storage that is reserved to satisfy the buffering requirements for one or more table spaces or indexes.

built-in data type. A data type that IBM supplies. Among the built-in data types for DB2 UDB for z/OS are string, numeric, ROWID, and datetime. Contrast with *distinct type*.

built-in function. A function that DB2 supplies. Contrast with *user-defined function*.

business dimension. A category of data, such as products or time periods, that an organization might want to analyze.

C

cache structure. A coupling facility structure that stores data that can be available to all members of a Sysplex. A DB2 data sharing group uses cache structures as group buffer pools.

CAF. Call attachment facility.

call attachment facility (CAF). A DB2 attachment facility for application programs that run in TSO or z/OS batch. The CAF is an alternative to the DSN command processor and provides greater control over the execution environment.

call-level interface (CLI). A callable application programming interface (API) for database access, which is an alternative to using embedded SQL. In contrast to embedded SQL, DB2 ODBC (which is based on the CLI architecture) does not require the user to precompile or bind applications, but instead provides a standard set of functions to process SQL statements and related services at run time.

cascade delete. The way in which DB2 enforces referential constraints when it deletes all descendent rows of a deleted parent row.

CASE expression. An expression that is selected based on the evaluation of one or more conditions.

cast function. A function that is used to convert instances of a (source) data type into instances of a different (target) data type. In general, a cast function has the name of the target data type. It has one single argument whose type is the source data type; its return type is the target data type.

castout. The DB2 process of writing changed pages from a group buffer pool to disk.

castout owner. The DB2 member that is responsible for casting out a particular page set or partition.

catalog. In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

catalog table. Any table in the DB2 catalog.

CCSID. Coded character set identifier.

CDB. Communications database.

CDRA. Character Data Representation Architecture.

CEC. Central electronic complex. See *central processor complex*.

central electronic complex (CEC). See *central processor complex*.

central processor (CP). The part of the computer that contains the sequencing and processing facilities for instruction execution, initial program load, and other machine operations.

central processor complex (CPC). A physical collection of hardware (such as an ES/3090™) that consists of main storage, one or more central processors, timers, and channels.

| **CFRM.** Coupling facility resource management.

CFRM policy. A declaration by a z/OS administrator regarding the allocation rules for a coupling facility structure.

character conversion. The process of changing characters from one encoding scheme to another.

Character Data Representation Architecture (CDRA). An architecture that is used to achieve consistent representation, processing, and interchange of string data.

character large object (CLOB). A sequence of bytes representing single-byte characters or a mixture of single- and double-byte characters where the size of the value can be up to 2 GB−1. In general, character large object values are used whenever a character string might exceed the limits of the VARCHAR type.

character set. A defined set of characters.

character string. A sequence of bytes that represent bit data, single-byte characters, or a mixture of single-byte and multibyte characters.

check constraint. A user-defined constraint that specifies the values that specific columns of a base table can contain.

check integrity. The condition that exists when each row in a table conforms to the check constraints that are defined on that table. Maintaining check integrity requires DB2 to enforce check constraints on operations that add or change data.

check pending. A state of a table space or partition that prevents its use by some utilities and by some SQL statements because of rows that violate referential constraints, check constraints, or both.

checkpoint. A point at which DB2 records internal status information on the DB2 log; the recovery process uses this information if DB2 abnormally terminates.

| **child lock.** For explicit hierarchical locking, a lock that
| is held on either a table, page, row, or a large object
| (LOB). Each child lock has a parent lock. See also
| *parent lock*.

CI. Control interval.

| **CICS.** Represents (in this publication): CICS
| Transaction Server for z/OS: Customer Information
| Control System Transaction Server for z/OS.

CICS attachment facility. A DB2 subcomponent that uses the z/OS subsystem interface (SSI) and cross-storage linkage to process requests from CICS to DB2 and to coordinate resource commitment.

CIDF. Control interval definition field.

claim. A notification to DB2 that an object is being accessed. Claims prevent drains from occurring until the claim is released, which usually occurs at a commit point. Contrast with *drain*.

claim class. A specific type of object access that can be one of the following isolation levels:
  Cursor stability (CS)
  Repeatable read (RR)
  Write

claim count. A count of the number of agents that are accessing an object.

class of service. A VTAM term for a list of routes through a network, arranged in an order of preference for their use.

class word. A single word that indicates the nature of a data attribute. For example, the class word PROJ indicates that the attribute identifies a project.

clause. In SQL, a distinct part of a statement, such as a SELECT clause or a WHERE clause.

CLI. Call- level interface.

client. See *requester*.

CLIST. Command list. A language for performing TSO tasks.

CLOB. Character large object.

closed application. An application that requires exclusive use of certain statements on certain DB2 objects, so that the objects are managed solely through the application's external interface.

CLPA. Create link pack area.

clustering index. An index that determines how rows are physically ordered (*clustered*) in a table space. If a clustering index on a partitioned table is not a partitioning index, the rows are ordered in cluster sequence within each data partition instead of spanning partitions. Prior to Version 8 of DB2 UDB for z/OS, the partitioning index was required to be the clustering index.

coded character set. A set of unambiguous rules that establish a character set and the one-to-one relationships between the characters of the set and their coded representations.

coded character set identifier (CCSID). A 16-bit number that uniquely identifies a coded representation of graphic characters. It designates an encoding scheme identifier and one or more pairs consisting of a character set identifier and an associated code page identifier.

code page. (1) A set of assignments of characters to code points. In EBCDIC, for example, the character 'A' is assigned code point X'C1' (2) , and character 'B' is assigned code point X'C2'. Within a code page, each code point has only one specific meaning.

code point. In CDRA, a unique bit pattern that represents a character in a code page.

coexistence. During migration, the period of time in which two releases exist in the same data sharing group.

cold start. A process by which DB2 restarts without processing any log records. Contrast with *warm start*.

collection. A group of packages that have the same qualifier.

column. The vertical component of a table. A column has a name and a particular data type (for example, character, decimal, or integer).

column function. An operation that derives its result by using values from one or more rows. Contrast with *scalar function*.

"come from" checking. An LU 6.2 security option that defines a list of authorization IDs that are allowed to connect to DB2 from a partner LU.

command. A DB2 operator command or a DSN subcommand. A command is distinct from an SQL statement.

command prefix. A one- to eight-character command identifier. The command prefix distinguishes the command as belonging to an application or subsystem rather than to MVS.

command recognition character (CRC). A character that permits a z/OS console operator or an IMS subsystem user to route DB2 commands to specific DB2 subsystems.

command scope. The scope of command operation in a data sharing group. If a command has *member scope*, the command displays information only from the one member or affects only non-shared resources that are owned locally by that member. If a command has *group scope*, the command displays information from all members, affects non-shared resources that are owned locally by all members, displays information on sharable resources, or affects sharable resources.

commit. The operation that ends a unit of work by releasing locks so that the database changes that are made by that unit of work can be perceived by other processes.

commit point. A point in time when data is considered consistent.

committed phase. The second phase of the multisite update process that requests all participants to commit the effects of the logical unit of work.

common service area (CSA). In z/OS, a part of the common area that contains data areas that are addressable by all address spaces.

communications database (CDB). A set of tables in the DB2 catalog that are used to establish conversations with remote database management systems.

comparison operator. A token (such as =, >, or <) that is used to specify a relationship between two values.

composite key. An ordered set of key columns of the same table.

compression dictionary. The dictionary that controls the process of compression and decompression. This dictionary is created from the data in the table space or table space partition.

concurrency. The shared use of resources by more than one application process at the same time.

conditional restart. A DB2 restart that is directed by a user-defined conditional restart control record (CRCR).

connection. In SNA, the existence of a communication path between two partner LUs that allows information to be exchanged (for example, two DB2 subsystems that are connected and communicating by way of a conversation).

connection context. In SQLJ, a Java™ object that represents a connection to a data source.

connection declaration clause. In SQLJ, a statement that declares a connection to a data source.

connection handle. The data object containing information that is associated with a connection that DB2 ODBC manages. This includes general status information, transaction status, and diagnostic information.

connection ID. An identifier that is supplied by the attachment facility and that is associated with a specific address space connection.

consistency token. A timestamp that is used to generate the version identifier for an application. See also *version*.

constant. A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. Contrast with *variable*.

constraint. A rule that limits the values that can be inserted, deleted, or updated in a table. See *referential constraint*, *check constraint*, and *unique constraint*.

context. The application's logical connection to the data source and associated internal DB2 ODBC connection information that allows the application to direct its operations to a data source. A DB2 ODBC context represents a DB2 thread.

contracting conversion. A process that occurs when the length of a converted string is smaller than that of the source string. For example, this process occurs when an EBCDIC mixed-data string that contains DBCS characters is converted to ASCII mixed data; the converted string is shorter because of the removal of the shift codes.

control interval (CI). A fixed-length area or disk in which VSAM stores records and creates distributed free space. Also, in a key-sequenced data set or file, the set of records that an entry in the sequence-set index record points to. The control interval is the unit of information that VSAM transmits to or from disk. A control interval always includes an integral number of physical records.

control interval definition field (CIDF). In VSAM, a field that is located in the 4 bytes at the end of each control interval; it describes the free space, if any, in the control interval.

conversation. Communication, which is based on LU 6.2 or Advanced Program-to-Program Communication (APPC), between an application and a remote transaction program over an SNA logical unit-to-logical unit (LU-LU) session that allows communication while processing a transaction.

coordinator. The system component that coordinates the commit or rollback of a unit of work that includes work that is done on one or more other systems.

copy pool. A named set of SMS storage groups that contains data that is to be copied collectively. A copy pool is an SMS construct that lets you define which storage groups are to be copied by using FlashCopy® functions. HSM determines which volumes belong to a copy pool.

copy target. A named set of SMS storage groups that are to be used as containers for copy pool volume copies. A copy target is an SMS construct that lets you define which storage groups are to be used as containers for volumes that are copied by using FlashCopy functions.

copy version. A point-in-time FlashCopy copy that is managed by HSM. Each copy pool has a version parameter that specifies how many copy versions are maintained on disk.

correlated columns. A relationship between the value of one column and the value of another column.

correlated subquery. A subquery (part of a WHERE or HAVING clause) that is applied to a row or group of rows of a table or view that is named in an outer subselect statement.

correlation ID. An identifier that is associated with a specific thread. In TSO, it is either an authorization ID or the job name.

correlation name. An identifier that designates a table, a view, or individual rows of a table or view within a single SQL statement. It can be defined in any FROM clause or in the first clause of an UPDATE or DELETE statement.

cost category. A category into which DB2 places cost estimates for SQL statements at the time the statement is bound. A cost estimate can be placed in either of the following cost categories:
- A: Indicates that DB2 had enough information to make a cost estimate without using default values.
- B: Indicates that some condition exists for which DB2 was forced to use default values for its estimate.

The cost category is externalized in the COST_CATEGORY column of the DSN_STATEMNT_TABLE when a statement is explained.

coupling facility. A special PR/SM™ LPAR logical partition that runs the coupling facility control program and provides high-speed caching, list processing, and locking functions in a Parallel Sysplex®.

coupling facility resource management. A component of z/OS that provides the services to manage coupling facility resources in a Parallel Sysplex. This management includes the enforcement of CFRM policies to ensure that the coupling facility and structure requirements are satisfied.

CP. Central processor.

CPC. Central processor complex.

C++ member. A data object or function in a structure, union, or class.

C++ member function. An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and to the member functions of objects in its class. Member functions are also called methods.

C++ object. (1) A region of storage. An object is created when a variable is defined or a new function is invoked. (2) An instance of a class.

CRC. Command recognition character.

CRCR. Conditional restart control record. See also *conditional restart*.

create link pack area (CLPA). An option that is used during IPL to initialize the link pack pageable area.

created temporary table. A table that holds temporary data and is defined with the SQL statement CREATE GLOBAL TEMPORARY TABLE. Information about created temporary tables is stored in the DB2 catalog, so this kind of table is persistent and can be shared across application processes. Contrast with *declared temporary table*. See also *temporary table*.

cross-memory linkage. A method for invoking a program in a different address space. The invocation is synchronous with respect to the caller.

cross-system coupling facility (XCF). A component of z/OS that provides functions to support cooperation between authorized programs that run within a Sysplex.

cross-system extended services (XES). A set of z/OS services that allow multiple instances of an application or subsystem, running on different systems in a Sysplex environment, to implement high-performance, high-availability data sharing by using a coupling facility.

CS. Cursor stability.

CSA. Common service area.

CT. Cursor table.

current data. Data within a host structure that is current with (identical to) the data within the base table.

current SQL ID. An ID that, at a single point in time, holds the privileges that are exercised when certain dynamic SQL statements run. The current SQL ID can be a primary authorization ID or a secondary authorization ID.

current status rebuild. The second phase of restart processing during which the status of the subsystem is reconstructed from information on the log.

cursor. A named control structure that an application program uses to point to a single row or multiple rows within some ordered set of rows of a result table. A cursor can be used to retrieve, update, or delete rows from a result table.

cursor sensitivity. The degree to which database updates are visible to the subsequent FETCH statements in a cursor. A cursor can be sensitive to changes that are made with positioned update and delete statements specifying the name of that cursor. A cursor can also be sensitive to changes that are made with searched update or delete statements, or with cursors other than this cursor. These changes can be made by this application process or by another application process.

cursor stability (CS). The isolation level that provides maximum concurrency without the ability to read uncommitted data. With cursor stability, a unit of work holds locks only on its uncommitted changes and on the current row of each of its cursors.

cursor table (CT). The copy of the skeleton cursor table that is used by an executing application process.

cycle. A set of tables that can be ordered so that each table is a descendent of the one before it, and the first table is a descendent of the last table. A self-referencing table is a cycle with a single member.

D

| **DAD.** See *Document access definition.*

| **disk.** A direct-access storage device that records data
| magnetically.

database. A collection of tables, or a collection of table spaces and index spaces.

database access thread. A thread that accesses data at the local subsystem on behalf of a remote subsystem.

database administrator (DBA). An individual who is responsible for designing, developing, operating, safeguarding, maintaining, and using a database.

| **database alias.** The name of the target server if
| different from the location name. The database alias
| name is used to provide the name of the database
| server as it is known to the network. When a database
| alias name is defined, the location name is used by the
| application to reference the server, but the database
| alias name is used to identify the database server to be
| accessed. Any fully qualified object names within any

SQL statements are not modified and are sent unchanged to the database server.

database descriptor (DBD). An internal representation of a DB2 database definition, which reflects the data definition that is in the DB2 catalog. The objects that are defined in a database descriptor are table spaces, tables, indexes, index spaces, relationships, check constraints, and triggers. A DBD also contains information about accessing tables in the database.

database exception status. An indication that something is wrong with a database. All members of a data sharing group must know and share the exception status of databases.

database identifier (DBID). An internal identifier of the database.

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and the access to the data that is stored within it.

database request module (DBRM). A data set member that is created by the DB2 precompiler and that contains information about SQL statements. DBRMs are used in the bind process.

database server. The target of a request from a local application or an intermediate database server. In the DB2 environment, the database server function is provided by the distributed data facility to access DB2 data from local applications, or from a remote database server that acts as an intermediate database server.

data currency. The state in which data that is retrieved into a host variable in your program is a copy of data in the base table.

data definition name (ddname). The name of a data definition (DD) statement that corresponds to a data control block containing the same name.

data dictionary. A repository of information about an organization's application programs, databases, logical data models, users, and authorizations. A data dictionary can be manual or automated.

data-driven business rules. Constraints on particular data values that exist as a result of requirements of the business.

Data Language/I (DL/I). The IMS data manipulation language; a common high-level interface between a user application and IMS.

data mart. A small data warehouse that applies to a single department or team. See also *data warehouse*.

data mining. The process of collecting critical business information from a data warehouse, correlating it, and uncovering associations, patterns, and trends.

data partition. A VSAM data set that is contained within a partitioned table space.

data-partitioned secondary index (DPSI). A secondary index that is partitioned. The index is partitioned according to the underlying data.

data sharing. The ability of two or more DB2 subsystems to directly access and change a single set of data.

data sharing group. A collection of one or more DB2 subsystems that directly access and change the same data while maintaining data integrity.

data sharing member. A DB2 subsystem that is assigned by XCF services to a data sharing group.

data source. A local or remote relational or non-relational data manager that is capable of supporting data access via an ODBC driver that supports the ODBC APIs. In the case of DB2 UDB for z/OS, the data sources are always relational database managers.

data space. In releases prior to DB2 UDB for z/OS, Version 8, a range of up to 2 GB of contiguous virtual storage addresses that a program can directly manipulate. Unlike an address space, a data space can hold only data; it does not contain common areas, system data, or programs.

data type. An attribute of columns, literals, host variables, special registers, and the results of functions and expressions.

data warehouse. A system that provides critical business information to an organization. The data warehouse system cleanses the data for accuracy and currency, and then presents the data to decision makers so that they can interpret and use it effectively and efficiently.

date. A three-part value that designates a day, month, and year.

date duration. A decimal integer that represents a number of years, months, and days.

datetime value. A value of the data type DATE, TIME, or TIMESTAMP.

DBA. Database administrator.

DBCLOB. Double-byte character large object.

DBCS. Double-byte character set.

DBD. Database descriptor.

DBID. Database identifier.

DBMS. Database management system.

DBRM. Database request module.

DB2 catalog. Tables that are maintained by DB2 and contain descriptions of DB2 objects, such as tables, views, and indexes.

DB2 command. An instruction to the DB2 subsystem that a user enters to start or stop DB2, to display information on current users, to start or stop databases, to display information on the status of databases, and so on.

DB2 for VSE & VM. The IBM DB2 relational database management system for the VSE and VM operating systems.

DB2I. DB2 Interactive.

DB2 Interactive (DB2I). The DB2 facility that provides for the execution of SQL statements, DB2 (operator) commands, programmer commands, and utility invocation.

DB2I Kanji Feature. The tape that contains the panels and jobs that allow a site to display DB2I panels in Kanji.

DB2 PM. DB2 Performance Monitor.

DB2 thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services.

DCLGEN. Declarations generator.

DDF. Distributed data facility.

ddname. Data definition name.

deadlock. Unresolvable contention for the use of a resource, such as a table or an index.

declarations generator (DCLGEN). A subcomponent of DB2 that generates SQL table declarations and COBOL, C, or PL/I data structure declarations that conform to the table. The declarations are generated from DB2 system catalog information. DCLGEN is also a DSN subcommand.

declared temporary table. A table that holds temporary data and is defined with the SQL statement DECLARE GLOBAL TEMPORARY TABLE. Information about declared temporary tables is not stored in the DB2 catalog, so this kind of table is not persistent and can be used only by the application process that issued the DECLARE statement. Contrast with *created temporary table*. See also *temporary table*.

default value. A predetermined value, attribute, or option that is assumed when no other is explicitly specified.

deferred embedded SQL. SQL statements that are neither fully static nor fully dynamic. Like static statements, they are embedded within an application, but like dynamic statements, they are prepared during the execution of the application.

deferred write. The process of asynchronously writing changed data pages to disk.

degree of parallelism. The number of concurrently executed operations that are initiated to process a query.

delete-connected. A table that is a dependent of table P or a dependent of a table to which delete operations from table P cascade.

delete hole. The location on which a cursor is positioned when a row in a result table is refetched and the row no longer exists on the base table, because another cursor deleted the row between the time the cursor first included the row in the result table and the time the cursor tried to refetch it.

delete rule. The rule that tells DB2 what to do to a dependent row when a parent row is deleted. For each relationship, the rule might be CASCADE, RESTRICT, SET NULL, or NO ACTION.

delete trigger. A trigger that is defined with the triggering SQL operation DELETE.

delimited identifier. A sequence of characters that are enclosed within double quotation marks ("). The sequence must consist of a letter followed by zero or more characters, each of which is a letter, digit, or the underscore character (_).

delimiter token. A string constant, a delimited identifier, an operator symbol, or any of the special characters that are shown in DB2 syntax diagrams.

denormalization. A key step in the task of building a physical relational database design. Denormalization is the intentional duplication of columns in multiple tables, and the consequence is increased data redundancy. Denormalization is sometimes necessary to minimize performance problems. Contrast with *normalization*.

dependent. An object (row, table, or table space) that has at least one parent. The object is also said to be a dependent (row, table, or table space) of its parent. See also *parent row*, *parent table*, *parent table space*.

dependent row. A row that contains a foreign key that matches the value of a primary key in the parent row.

dependent table. A table that is a dependent in at least one referential constraint.

DES-based authenticator. An authenticator that is generated using the DES algorithm.

descendent. An object that is a dependent of an object or is the dependent of a descendent of an object.

descendent row. A row that is dependent on another row, or a row that is a descendent of a dependent row.

descendent table. A table that is a dependent of another table, or a table that is a descendent of a dependent table.

deterministic function. A user-defined function whose result is dependent on the values of the input arguments. That is, successive invocations with the same input values produce the same answer. Sometimes referred to as a *not-variant* function. Contrast this with an *nondeterministic function* (sometimes called a *variant function*), which might not always produce the same result for the same inputs.

DFP. Data Facility Product (in z/OS).

DFSMS. Data Facility Storage Management Subsystem (in z/OS). Also called *Storage Management Subsystem (SMS)*.

| **DFSMSdss™.** The data set services (dss) component
| of DFSMS (in z/OS).

| **DFSMShsm™.** The hierarchical storage manager
| (hsm) component of DFSMS (in z/OS).

dimension. A data category such as time, products, or markets. The elements of a dimension are referred to as members. Dimensions offer a very concise, intuitive way of organizing and selecting data for retrieval, exploration, and analysis. See also *dimension table*.

dimension table. The representation of a dimension in a star schema. Each row in a dimension table represents all of the attributes for a particular member of the dimension. See also *dimension*, *star schema*, and *star join*.

directory. The DB2 system database that contains internal objects such as database descriptors and skeleton cursor tables.

distinct type. A user-defined data type that is internally represented as an existing type (its source type), but is considered to be a separate and incompatible type for semantic purposes.

distributed data. Data that resides on a DBMS other than the local system.

distributed data facility (DDF). A set of DB2 components through which DB2 communicates with another relational database management system.

Distributed Relational Database Architecture™ (DRDA). A connection protocol for distributed relational database processing that is used by IBM's relational database products. DRDA includes protocols for communication between an application and a remote relational database management system, and for communication between relational database management systems. See also *DRDA access.*

DL/I. Data Language/I.

DNS. Domain name server.

| **document access definition (DAD).** Used to define
| the indexing scheme for an XML column or the mapping
| scheme of an XML collection. It can be used to enable
| an XML Extender column of an XML collection, which is
| XML formatted.

domain. The set of valid values for an attribute.

domain name. The name by which TCP/IP applications refer to a TCP/IP host within a TCP/IP network.

domain name server (DNS). A special TCP/IP network server that manages a distributed directory that is used to map TCP/IP host names to IP addresses.

double-byte character large object (DBCLOB). A sequence of bytes representing double-byte characters where the size of the values can be up to 2 GB. In general, DBCLOB values are used whenever a double-byte character string might exceed the limits of the VARGRAPHIC type.

double-byte character set (DBCS). A set of characters, which are used by national languages such as Japanese and Chinese, that have more symbols than can be represented by a single byte. Each character is 2 bytes in length. Contrast with *single-byte character set* and *multibyte character set*.

double-precision floating point number. A 64-bit approximate representation of a real number.

downstream. The set of nodes in the syncpoint tree that is connected to the local DBMS as a participant in the execution of a two-phase commit.

| **DPSI.** Data-partitioned secondary index.

drain. The act of acquiring a locked resource by quiescing access to that object.

drain lock. A lock on a claim class that prevents a claim from occurring.

DRDA. Distributed Relational Database Architecture.

DRDA access. An open method of accessing distributed data that you can use to can connect to another database server to execute packages that were previously bound at the server location. You use the

SQL CONNECT statement or an SQL statement with a three-part name to identify the server. Contrast with *private protocol access*.

DSN. (1) The default DB2 subsystem name. (2) The name of the TSO command processor of DB2. (3) The first three characters of DB2 module and macro names.

duration. A number that represents an interval of time. See also *date duration*, *labeled duration*, and *time duration*.

| **dynamic cursor.** A named control structure that an application program uses to change the size of the result table and the order of its rows after the cursor is opened. Contrast with *static cursor.*

dynamic dump. A dump that is issued during the execution of a program, usually under the control of that program.

dynamic SQL. SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution.

| **dynamic statement cache pool.** A cache, located above the 2-GB storage line, that holds dynamic statements.

E

EA-enabled table space. A table space or index space that is enabled for extended addressability and that contains individual partitions (or pieces, for LOB table spaces) that are greater than 4 GB.

EB. See *exabyte*.

EBCDIC. Extended binary coded decimal interchange code. An encoding scheme that is used to represent character data in the z/OS, VM, VSE, and iSeries™ environments. Contrast with *ASCII* and *Unicode*.

e-business. The transformation of key business processes through the use of Internet technologies.

| **EDM pool.** A pool of main storage that is used for database descriptors, application plans, authorization cache, application packages.

EID. Event identifier.

embedded SQL. SQL statements that are coded within an application program. See *static SQL*.

enclave. In Language Environment , an independent collection of routines, one of which is designated as the main routine. An enclave is similar to a program or run unit.

encoding scheme. A set of rules to represent character data (ASCII, EBCDIC, or Unicode).

entity. A significant object of interest to an organization.

enumerated list. A set of DB2 objects that are defined with a LISTDEF utility control statement in which pattern-matching characters (*, %, _ or ?) are not used.

environment. A collection of names of logical and physical resources that are used to support the performance of a function.

environment handle. In DB2 ODBC, the data object that contains global information regarding the state of the application. An environment handle must be allocated before a connection handle can be allocated. Only one environment handle can be allocated per application.

EOM. End of memory.

EOT. End of task.

equijoin. A join operation in which the join-condition has the form *expression* = *expression*.

error page range. A range of pages that are considered to be physically damaged. DB2 does not allow users to access any pages that fall within this range.

escape character. The symbol that is used to enclose an SQL delimited identifier. The escape character is the double quotation mark ("), except in COBOL applications, where the user assigns the symbol, which is either a double quotation mark or an apostrophe (').

ESDS. Entry sequenced data set.

ESMT. External subsystem module table (in IMS).

EUR. IBM European Standards.

exabyte. For processor, real and virtual storage capacities and channel volume:
1 152 921 504 606 846 976 bytes or 2^{60}.

exception table. A table that holds rows that violate referential constraints or check constraints that the CHECK DATA utility finds.

exclusive lock. A lock that prevents concurrently executing application processes from reading or changing data. Contrast with *share lock*.

executable statement. An SQL statement that can be embedded in an application program, dynamically prepared and executed, or issued interactively.

execution context. In SQLJ, a Java object that can be used to control the execution of SQL statements.

exit routine. A user-written (or IBM-provided default) program that receives control from DB2 to perform specific functions. Exit routines run as extensions of DB2.

expanding conversion. A process that occurs when the length of a converted string is greater than that of the source string. For example, this process occurs when an ASCII mixed-data string that contains DBCS characters is converted to an EBCDIC mixed-data string; the converted string is longer because of the addition of shift codes.

explicit hierarchical locking. Locking that is used to make the parent-child relationship between resources known to IRLM. This kind of locking avoids global locking overhead when no inter-DB2 interest exists on a resource.

exposed name. A correlation name or a table or view name for which a correlation name is not specified. Names that are specified in a FROM clause are exposed or non-exposed.

expression. An operand or a collection of operators and operands that yields a single value.

extended recovery facility (XRF). A facility that minimizes the effect of failures in z/OS, VTAM , the host processor, or high-availability applications during sessions between high-availability applications and designated terminals. This facility provides an alternative subsystem to take over sessions from the failing subsystem.

Extensible Markup Language (XML). A standard metalanguage for defining markup languages that is a subset of Standardized General Markup Language (SGML). The less complex nature of XML makes it easier to write applications that handle document types, to author and manage structured information, and to transmit and share structured information across diverse computing environments.

external function. A function for which the body is written in a programming language that takes scalar argument values and produces a scalar result for each invocation. Contrast with *sourced function*, *built-in function*, and *SQL function*.

external procedure. A user-written application program that can be invoked with the SQL CALL statement, which is written in a programming language. Contrast with *SQL procedure*.

external routine. A user-defined function or stored procedure that is based on code that is written in an external programming language.

external subsystem module table (ESMT). In IMS, the table that specifies which attachment modules must be loaded.

F

failed member state. A state of a member of a data sharing group. When a member fails, the XCF permanently records the failed member state. This state usually means that the member's task, address space, or z/OS system terminated before the state changed from active to quiesced.

fallback. The process of returning to a previous release of DB2 after attempting or completing migration to a current release.

false global lock contention. A contention indication from the coupling facility when multiple lock names are hashed to the same indicator and when no real contention exists.

fan set. A direct physical access path to data, which is provided by an index, hash, or link; a fan set is the means by which the data manager supports the ordering of data.

federated database. The combination of a DB2 Universal Database server (in Linux, UNIX, and Windows environments) and multiple data sources to which the server sends queries. In a federated database system, a client application can use a single SQL statement to join data that is distributed across multiple database management systems and can view the data as if it were local.

fetch orientation. The specification of the desired placement of the cursor as part of a FETCH statement (for example, BEFORE, AFTER, NEXT, PRIOR, CURRENT, FIRST, LAST, ABSOLUTE, and RELATIVE).

field procedure. A user-written exit routine that is designed to receive a single value and transform (encode or decode) it in any way the user can specify.

filter factor. A number between zero and one that estimates the proportion of rows in a table for which a predicate is true.

fixed-length string. A character or graphic string whose length is specified and cannot be changed. Contrast with *varying-length string*.

FlashCopy. A function on the IBM Enterprise Storage Server® that can create a point-in-time copy of data while an application is running.

foreign key. A column or set of columns in a dependent table of a constraint relationship. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table. Each foreign key value must either match a parent key value in the related parent table or be null.

| **forest.** An ordered set of subtrees of XML nodes.

forget. In a two-phase commit operation, (1) the vote that is sent to the prepare phase when the participant has not modified any data. The forget vote allows a participant to release locks and forget about the logical unit of work. This is also referred to as the read-only vote. (2) The response to the *committed* request in the second phase of the operation.

forward log recovery. The third phase of restart processing during which DB2 processes the log in a forward direction to apply all REDO log records.

free space. The total amount of unused space in a page; that is, the space that is not used to store records or control information is free space.

full outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of both tables. See also *join*.

fullselect. A subselect, a values-clause, or a number of both that are combined by set operators. *Fullselect* specifies a result table. If UNION is not used, the result of the fullselect is the result of the specified subselect.

| **fully escaped mapping.** A mapping from an SQL
| identifier to an XML name when the SQL identifier is a
| column name.

function. A mapping, which is embodied as a program (the function body) that is invocable by means of zero or more input values (arguments) to a single value (the result). See also *column function* and *scalar function*.

Functions can be user-defined, built-in, or generated by DB2. (See also *built-in function*, *cast function*, *external function*, *sourced function*, *SQL function*, and *user-defined function*.)

function definer. The authorization ID of the owner of the schema of the function that is specified in the CREATE FUNCTION statement.

function implementer. The authorization ID of the owner of the function program and function package.

function package. A package that results from binding the DBRM for a function program.

function package owner. The authorization ID of the user who binds the function program's DBRM into a function package.

function resolution. The process, internal to the DBMS, by which a function invocation is bound to a particular function instance. This process uses the function name, the data types of the arguments, and a list of the applicable schema names (called the *SQL path*) to make the selection. This process is sometimes called *function selection*.

function selection. See *function resolution*.

function signature. The logical concatenation of a fully qualified function name with the data types of all of its parameters.

G

GB. Gigabyte (1 073 741 824 bytes).

GBP. Group buffer pool.

GBP-dependent. The status of a page set or page set partition that is dependent on the group buffer pool. Either read/write interest is active among DB2 subsystems for this page set, or the page set has changed pages in the group buffer pool that have not yet been cast out to disk.

generalized trace facility (GTF). A z/OS service program that records significant system events such as I/O interrupts, SVC interrupts, program interrupts, or external interrupts.

generic resource name. A name that VTAM uses to represent several application programs that provide the same function in order to handle session distribution and balancing in a Sysplex environment.

getpage. An operation in which DB2 accesses a data page.

global lock. A lock that provides concurrency control within and among DB2 subsystems. The scope of the lock is across all DB2 subsystems of a data sharing group.

global lock contention. Conflicts on locking requests between different DB2 members of a data sharing group when those members are trying to serialize shared resources.

governor. See *resource limit facility*.

graphic string. A sequence of DBCS characters.

gross lock. The *shared*, *update*, or *exclusive* mode locks on a table, partition, or table space.

group buffer pool (GBP). A coupling facility cache structure that is used by a data sharing group to cache data and to ensure that the data is consistent for all members.

group buffer pool duplexing. The ability to write data to two instances of a group buffer pool structure: a *primary group buffer pool* and a *secondary group buffer pool*. z/OS publications refer to these instances as the "old" (for primary) and "new" (for secondary) structures.

group level. The release level of a data sharing group, which is established when the first member migrates to a new release.

group name. The z/OS XCF identifier for a data sharing group.

group restart. A restart of at least one member of a data sharing group after the loss of either locks or the shared communications area.

GTF. Generalized trace facility.

H

handle. In DB2 ODBC, a variable that refers to a data structure and associated resources. See also *statement handle*, *connection handle*, and *environment handle*.

help panel. A screen of information that presents tutorial text to assist a user at the workstation or terminal.

heuristic damage. The inconsistency in data between one or more participants that results when a heuristic decision to resolve an indoubt LUW at one or more participants differs from the decision that is recorded at the coordinator.

heuristic decision. A decision that forces indoubt resolution at a participant by means other than automatic resynchronization between coordinator and participant.

hole. A row of the result table that cannot be accessed because of a delete or an update that has been performed on the row. See also *delete hole* and *update hole*.

home address space. The area of storage that z/OS currently recognizes as *dispatched*.

host. The set of programs and resources that are available on a given TCP/IP instance.

host expression. A Java variable or expression that is referenced by SQL clauses in an SQLJ application program.

host identifier. A name that is declared in the host program.

host language. A programming language in which you can embed SQL statements.

host program. An application program that is written in a host language and that contains embedded SQL statements.

host structure. In an application program, a structure that is referenced by embedded SQL statements.

host variable. In an application program, an application variable that is referenced by embedded SQL statements.

| **host variable array.** An array of elements, each of
| which corresponds to a value for a column. The
| dimension of the array determines the maximum
| number of rows for which the array can be used.

HSM. Hierarchical storage manager.

HTML. Hypertext Markup Language, a standard method for presenting Web data to users.

HTTP. Hypertext Transfer Protocol, a communication protocol that the Web uses.

I

ICF. Integrated catalog facility.

IDCAMS. An IBM program that is used to process access method services commands. It can be invoked as a job or jobstep, from a TSO terminal, or from within a user's application program.

IDCAMS LISTCAT. A facility for obtaining information that is contained in the access method services catalog.

identify. A request that an attachment service program in an address space that is separate from DB2 issues thorough the z/OS subsystem interface to inform DB2 of its existence and to initiate the process of becoming connected to DB2.

identity column. A column that provides a way for DB2 to automatically generate a numeric value for each row. The generated values are unique if cycling is not used. Identity columns are defined with the AS IDENTITY clause. Uniqueness of values can be ensured by defining a unique index that contains only the identity column. A table can have no more than one identity column.

IFCID. Instrumentation facility component identifier.

IFI. Instrumentation facility interface.

IFI call. An invocation of the instrumentation facility interface (IFI) by means of one of its defined functions.

IFP. IMS Fast Path.

image copy. An exact reproduction of all or part of a table space. DB2 provides utility programs to make full image copies (to copy the entire table space) or incremental image copies (to copy only those pages that have been modified since the last image copy).

implied forget. In the presumed-abort protocol, an implied response of *forget* to the second-phase *committed* request from the coordinator. The response is implied when the participant responds to any subsequent request from the coordinator.

IMS. Information Management System.

IMS attachment facility. A DB2 subcomponent that uses z/OS subsystem interface (SSI) protocols and cross-memory linkage to process requests from IMS to DB2 and to coordinate resource commitment.

IMS DB. Information Management System Database.

IMS TM. Information Management System Transaction Manager.

in-abort. A status of a unit of recovery. If DB2 fails after a unit of recovery begins to be rolled back, but before the process is completed, DB2 continues to back out the changes during restart.

in-commit. A status of a unit of recovery. If DB2 fails after beginning its phase 2 commit processing, it "knows," when restarted, that changes made to data are consistent. Such units of recovery are termed *in-commit*.

independent. An object (row, table, or table space) that is neither a parent nor a dependent of another object.

index. A set of pointers that are logically ordered by the values of a key. Indexes can provide faster access to data and can enforce uniqueness on the rows in a table.

| **index-controlled partitioning.** A type of partitioning in
| which partition boundaries for a partitioned table are
| controlled by values that are specified on the CREATE
| INDEX statement. Partition limits are saved in the
| LIMITKEY column of the SYSIBM.SYSINDEXPART
| catalog table.

index key. The set of columns in a table that is used to determine the order of index entries.

index partition. A VSAM data set that is contained within a partitioning index space.

index space. A page set that is used to store the entries of one index.

indicator column. A 4-byte value that is stored in a base table in place of a LOB column.

indicator variable. A variable that is used to represent the null value in an application program. If the value for the selected column is null, a negative value is placed in the indicator variable.

indoubt. A status of a unit of recovery. If DB2 fails after it has finished its phase 1 commit processing and before it has started phase 2, only the commit coordinator knows if an individual unit of recovery is to be committed or rolled back. At emergency restart, if DB2 lacks the information it needs to make this decision, the status of the unit of recovery is *indoubt* until DB2 obtains this information from the coordinator. More than one unit of recovery can be indoubt at restart.

indoubt resolution. The process of resolving the status of an indoubt logical unit of work to either the committed or the rollback state.

inflight. A status of a unit of recovery. If DB2 fails before its unit of recovery completes phase 1 of the commit process, it merely backs out the updates of its unit of recovery at restart. These units of recovery are termed *inflight*.

inheritance. The passing downstream of class resources or attributes from a parent class in the class hierarchy to a child class.

initialization file. For DB2 ODBC applications, a file containing values that can be set to adjust the performance of the database manager.

inline copy. A copy that is produced by the LOAD or REORG utility. The data set that the inline copy produces is logically equivalent to a full image copy that is produced by running the COPY utility with read-only access (SHRLEVEL REFERENCE).

inner join. The result of a join operation that includes only the matched rows of both tables that are being joined. See also *join*.

inoperative package. A package that cannot be used because one or more user-defined functions or procedures that the package depends on were dropped. Such a package must be explicitly rebound. Contrast with *invalid package.*

| **insensitive cursor.** A cursor that is not sensitive to
| inserts, updates, or deletes that are made to the
| underlying rows of a result table after the result table
| has been materialized.

insert trigger. A trigger that is defined with the triggering SQL operation INSERT.

install. The process of preparing a DB2 subsystem to operate as a z/OS subsystem.

installation verification scenario. A sequence of operations that exercises the main DB2 functions and tests whether DB2 was correctly installed.

instrumentation facility component identifier (IFCID). A value that names and identifies a trace record of an event that can be traced. As a parameter on the START TRACE and MODIFY TRACE commands, it specifies that the corresponding event is to be traced.

instrumentation facility interface (IFI). A programming interface that enables programs to obtain online trace data about DB2, to submit DB2 commands, and to pass data to DB2.

Interactive System Productivity Facility (ISPF). An IBM licensed program that provides interactive dialog services in a z/OS environment.

inter-DB2 R/W interest. A property of data in a table space, index, or partition that has been opened by more than one member of a data sharing group and that has been opened for writing by at least one of those members.

intermediate database server. The target of a request from a local application or a remote application requester that is forwarded to another database server. In the DB2 environment, the remote request is forwarded transparently to another database server if the object that is referenced by a three-part name does not reference the local location.

internationalization. The support for an encoding scheme that is able to represent the code points of characters from many different geographies and languages. To support all geographies, the Unicode standard requires more than 1 byte to represent a single character. See also *Unicode*.

internal resource lock manager (IRLM). A z/OS subsystem that DB2 uses to control communication and database locking.

| **International Organization for Standardization.** An
| international body charged with creating standards to
| facilitate the exchange of goods and services as well as
| cooperation in intellectual, scientific, technological, and
| economic activity.

invalid package. A package that depends on an object (other than a user-defined function) that is dropped. Such a package is implicitly rebound on invocation. Contrast with *inoperative package.*

invariant character set. (1) A character set, such as the syntactic character set, whose code point assignments do not change from code page to code page. (2) A minimum set of characters that is available as part of all character sets.

IP address. A 4-byte value that uniquely identifies a TCP/IP host.

IRLM. Internal resource lock manager.

ISO. International Organization for Standardization.

isolation level. The degree to which a unit of work is isolated from the updating operations of other units of work. See also *cursor stability*, *read stability*, *repeatable read*, and *uncommitted read*.

ISPF. Interactive System Productivity Facility.

ISPF/PDF. Interactive System Productivity Facility/Program Development Facility.

iterator. In SQLJ, an object that contains the result set of a query. An iterator is equivalent to a cursor in other host languages.

iterator declaration clause. In SQLJ, a statement that generates an iterator declaration class. An iterator is an object of an iterator declaration class.

J

| **Japanese Industrial Standard.** An encoding scheme
| that is used to process Japanese characters.

| **JAR.** Java Archive.

Java Archive (JAR). A file format that is used for aggregating many files into a single file.

JCL. Job control language.

JDBC. A Sun Microsystems database application programming interface (API) for Java that allows programs to access database management systems by using callable SQL. JDBC does not require the use of an SQL preprocessor. In addition, JDBC provides an architecture that lets users add modules called *database drivers*, which link the application to their choice of database management systems at run time.

JES. Job Entry Subsystem.

JIS. Japanese Industrial Standard.

job control language (JCL). A control language that is used to identify a job to an operating system and to describe the job's requirements.

Job Entry Subsystem (JES). An IBM licensed program that receives jobs into the system and processes all output data that is produced by the jobs.

join. A relational operation that allows retrieval of data from two or more tables based on matching column values. See also *equijoin, full outer join, inner join, left outer join, outer join, and right outer join*.

K

KB. Kilobyte (1024 bytes).

Kerberos. A network authentication protocol that is designed to provide strong authentication for client/server applications by using secret-key cryptography.

Kerberos ticket. A transparent application mechanism that transmits the identity of an initiating principal to its target. A simple ticket contains the principal's identity, a session key, a timestamp, and other information, which is sealed using the target's secret key.

key. A column or an ordered collection of columns that is identified in the description of a table, index, or referential constraint. The same column can be part of more than one key.

key-sequenced data set (KSDS). A VSAM file or data set whose records are loaded in key sequence and controlled by an index.

keyword. In SQL, a name that identifies an option that is used in an SQL statement.

KSDS. Key-sequenced data set.

L

labeled duration. A number that represents a duration of years, months, days, hours, minutes, seconds, or microseconds.

large object (LOB). A sequence of bytes representing bit data, single-byte characters, double-byte characters, or a mixture of single- and double-byte characters. A LOB can be up to 2 GB−1 byte in length. See also *BLOB*, *CLOB*, and *DBCLOB*.

last agent optimization. An optimized commit flow for either presumed-nothing or presumed-abort protocols in which the last agent, or final participant, becomes the commit coordinator. This flow saves at least one message.

latch. A DB2 internal mechanism for controlling concurrent events or the use of system resources.

LCID. Log control interval definition.

LDS. Linear data set.

leaf page. A page that contains pairs of keys and RIDs and that points to actual data. Contrast with *nonleaf page*.

left outer join. The result of a join operation that includes the matched rows of both tables that are being joined, and that preserves the unmatched rows of the first table. See also *join*.

limit key. The highest value of the index key for a partition.

linear data set (LDS). A VSAM data set that contains data but no control information. A linear data set can be accessed as a byte-addressable string in virtual storage.

linkage editor. A computer program for creating load modules from one or more object modules or load modules by resolving cross references among the modules and, if necessary, adjusting addresses.

link-edit. The action of creating a loadable computer program using a linkage editor.

list. A type of object, which DB2 utilities can process, that identifies multiple table spaces, multiple index spaces, or both. A list is defined with the LISTDEF utility control statement.

list structure. A coupling facility structure that lets data be shared and manipulated as elements of a queue.

LLE. Load list element.

L-lock. Logical lock.

| **load list element.** A z/OS control block that controls
| the loading and deleting of a particular load module
| based on entry point names.

load module. A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

LOB. Large object.

LOB locator. A mechanism that allows an application program to manipulate a large object value in the database system. A LOB locator is a fullword integer value that represents a single LOB value. An application program retrieves a LOB locator into a host variable and can then apply SQL operations to the associated LOB value using the locator.

LOB lock. A lock on a LOB value.

LOB table space. A table space in an auxiliary table that contains all the data for a particular LOB column in the related base table.

local. A way of referring to any object that the local DB2 subsystem maintains. A *local table*, for example, is a table that is maintained by the local DB2 subsystem. Contrast with *remote*.

locale. The definition of a subset of a user's environment that combines a CCSID and characters that are defined for a specific language and country.

local lock. A lock that provides intra-DB2 concurrency control, but not inter-DB2 concurrency control; that is, its scope is a single DB2.

local subsystem. The unique relational DBMS to which the user or application program is directly connected (in the case of DB2, by one of the DB2 attachment facilities).

| **location.** The unique name of a database server. An
| application uses the location name to access a DB2
| database server. A database alias can be used to
| override the location name when accessing a remote
| server.

| **location alias.** Another name by which a database
| server identifies itself in the network. Applications can
| use this name to access a DB2 database server.

lock. A means of controlling concurrent events or access to data. DB2 locking is performed by the IRLM.

lock duration. The interval over which a DB2 lock is held.

lock escalation. The promotion of a lock from a row, page, or LOB lock to a table space lock because the number of page locks that are concurrently held on a given resource exceeds a preset limit.

locking. The process by which the integrity of data is ensured. Locking prevents concurrent users from accessing inconsistent data.

lock mode. A representation for the type of access that concurrently running programs can have to a resource that a DB2 lock is holding.

lock object. The resource that is controlled by a DB2 lock.

lock promotion. The process of changing the size or mode of a DB2 lock to a higher, more restrictive level.

lock size. The amount of data that is controlled by a DB2 lock on table data; the value can be a row, a page, a LOB, a partition, a table, or a table space.

lock structure. A coupling facility data structure that is composed of a series of lock entries to support shared and exclusive locking for logical resources.

log. A collection of records that describe the events that occur during DB2 execution and that indicate their sequence. The information thus recorded is used for recovery in the event of a failure during DB2 execution.

| **log control interval definition.** A suffix of the physical
| log record that tells how record segments are placed in
| the physical control interval.

logical claim. A claim on a logical partition of a nonpartitioning index.

logical data modeling. The process of documenting the comprehensive business information requirements in an accurate and consistent format. Data modeling is the first task of designing a database.

logical drain. A drain on a logical partition of a nonpartitioning index.

logical index partition. The set of all keys that reference the same data partition.

logical lock (L-lock). The lock type that transactions use to control intra- and inter-DB2 data concurrency between transactions. Contrast with *physical lock (P-lock)*.

logically complete. A state in which the concurrent copy process is finished with the initialization of the target objects that are being copied. The target objects are available for update.

logical page list (LPL). A list of pages that are in error and that cannot be referenced by applications until the pages are recovered. The page is in *logical error* because the actual media (coupling facility or disk) might not contain any errors. Usually a connection to the media has been lost.

logical partition. A set of key or RID pairs in a nonpartitioning index that are associated with a particular partition.

logical recovery pending (LRECP). The state in which the data and the index keys that reference the data are inconsistent.

logical unit (LU). An access point through which an application program accesses the SNA network in order to communicate with another application program.

logical unit of work (LUW). The processing that a program performs between synchronization points.

logical unit of work identifier (LUWID). A name that uniquely identifies a thread within a network. This name consists of a fully-qualified LU network name, an LUW instance number, and an LUW sequence number.

log initialization. The first phase of restart processing during which DB2 attempts to locate the current end of the log.

log record header (LRH). A prefix, in every logical record, that contains control information.

log record sequence number (LRSN). A unique identifier for a log record that is associated with a data sharing member. DB2 uses the LRSN for recovery in the data sharing environment.

log truncation. A process by which an explicit starting RBA is established. This RBA is the point at which the next byte of log data is to be written.

LPL. Logical page list.

LRECP. Logical recovery pending.

LRH. Log record header.

LRSN. Log record sequence number.

LU. Logical unit.

LU name. Logical unit name, which is the name by which VTAM refers to a node in a network. Contrast with *location name*.

LUW. Logical unit of work.

LUWID. Logical unit of work identifier.

M

mapping table. A table that the REORG utility uses to map the associations of the RIDs of data records in the original copy and in the shadow copy. This table is created by the user.

mass delete. The deletion of all rows of a table.

master terminal. The IMS logical terminal that has complete control of IMS resources during online operations.

master terminal operator (MTO). See *master terminal.*

materialize. (1) The process of putting rows from a view or nested table expression into a work file for additional processing by a query.

(2) The placement of a LOB value into contiguous storage. Because LOB values can be very large, DB2 avoids materializing LOB data until doing so becomes absolutely necessary.

| **materialized query table.** A table that is used to
| contain information that is derived and can be
| summarized from one or more source tables.

MB. Megabyte (1 048 576 bytes).

MBCS. Multibyte character set. UTF-8 is an example of an MBCS. Characters in UTF-8 can range from 1 to 4 bytes in DB2.

member name. The z/OS XCF identifier for a particular DB2 subsystem in a data sharing group.

menu. A displayed list of available functions for selection by the operator. A menu is sometimes called a *menu panel*.

metalanguage. A language that is used to create other specialized languages.

migration. The process of converting a subsystem with a previous release of DB2 to an updated or current release. In this process, you can acquire the functions of the updated or current release without losing the data that you created on the previous release.

mixed data string. A character string that can contain both single-byte and double-byte characters.

MLPA. Modified link pack area.

MODEENT. A VTAM macro instruction that associates a logon mode name with a set of parameters representing session protocols. A set of MODEENT macro instructions defines a logon mode table.

modeling database. A DB2 database that you create on your workstation that you use to model a DB2 UDB for z/OS subsystem, which can then be evaluated by the Index Advisor.

mode name. A VTAM name for the collection of physical and logical characteristics and attributes of a session.

modify locks. An L-lock or P-lock with a MODIFY attribute. A list of these active locks is kept at all times in the coupling facility lock structure. If the requesting DB2 subsystem fails, that DB2 subsystem's modify locks are converted to retained locks.

MPP. Message processing program (in IMS).

MTO. Master terminal operator.

multibyte character set (MBCS). A character set that represents single characters with more than a single byte. Contrast with *single-byte character set* and *double-byte character set*. See also *Unicode*.

multidimensional analysis. The process of assessing and evaluating an enterprise on more than one level.

Multiple Virtual Storage. An element of the z/OS operating system. This element is also called the Base Control Program (BCP).

multisite update. Distributed relational database processing in which data is updated in more than one location within a single unit of work.

multithreading. Multiple TCBs that are executing one copy of DB2 ODBC code concurrently (sharing a processor) or in parallel (on separate central processors).

must-complete. A state during DB2 processing in which the entire operation must be completed to maintain data integrity.

mutex. Pthread mutual exclusion; a lock. A Pthread mutex variable is used as a locking mechanism to allow serialization of critical sections of code by temporarily blocking the execution of all but one thread.

| **MVS.** See *Multiple Virtual Storage*.

N

negotiable lock. A lock whose mode can be downgraded, by agreement among contending users, to be compatible to all. A physical lock is an example of a negotiable lock.

nested table expression. A fullselect in a FROM clause (surrounded by parentheses).

network identifier (NID). The network ID that is assigned by IMS or CICS, or if the connection type is RRSAF, the RRS unit of recovery ID (URID).

NID. Network identifier.

nonleaf page. A page that contains keys and page numbers of other pages in the index (either leaf or nonleaf pages). Nonleaf pages never point to actual data.

| **nonpartitioned index.** An index that is not physically partitioned. Both partitioning indexes and secondary indexes can be nonpartitioned.

nonscrollable cursor. A cursor that can be moved only in a forward direction. Nonscrollable cursors are sometimes called forward-only cursors or serial cursors.

normalization. A key step in the task of building a logical relational database design. Normalization helps you avoid redundancies and inconsistencies in your data. An entity is normalized if it meets a set of constraints for a particular normal form (first normal form, second normal form, and so on). Contrast with *denormalization*.

nondeterministic function. A user-defined function whose result is not solely dependent on the values of the input arguments. That is, successive invocations with the same argument values can produce a different answer. this type of function is sometimes called a *variant* function. Contrast this with a *deterministic function* (sometimes called a *not-variant function*), which always produces the same result for the same inputs.

not-variant function. See *deterministic function*.

| **NPSI.** See *nonpartitioned secondary index.*

NRE. Network recovery element.

NUL. The null character ('\0'), which is represented by the value X'00'. In C, this character denotes the end of a string.

null. A special value that indicates the absence of information.

NULLIF. A scalar function that evaluates two passed expressions, returning either NULL if the arguments are equal or the value of the first argument if they are not.

null-terminated host variable. A varying-length host variable in which the end of the data is indicated by a null terminator.

null terminator. In C, the value that indicates the end of a string. For EBCDIC, ASCII, and Unicode UTF-8 strings, the null terminator is a single-byte value (X'00'). For Unicode UCS-2 (wide) strings, the null terminator is a double-byte value (X'0000').

O

OASN (origin application schedule number). In IMS, a 4-byte number that is assigned sequentially to each IMS schedule since the last cold start of IMS. The OASN is used as an identifier for a unit of work. In an 8-byte format, the first 4 bytes contain the schedule number and the last 4 bytes contain the number of IMS sync points (*commit points*) during the current schedule. The OASN is part of the NID for an IMS connection.

ODBC. Open Database Connectivity.

ODBC driver. A dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

OBID. Data object identifier.

Open Database Connectivity (ODBC). A Microsoft database application programming interface (API) for C that allows access to database management systems by using callable SQL. ODBC does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that lets users add modules called *database drivers*, which link the application to their choice of database management systems at run time. This means that applications no longer need to be directly linked to the modules of all the database management systems that are supported.

ordinary identifier. An uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier must not be a reserved word.

ordinary token. A numeric constant, an ordinary identifier, a host identifier, or a keyword.

originating task. In a parallel group, the primary agent that receives data from other execution units (referred to as *parallel tasks*) that are executing portions of the query in parallel.

OS/390. Operating System/390®.

OS/390 OpenEdition® Distributed Computing Environment (OS/390 OE DCE). A set of technologies that are provided by the Open Software Foundation to implement distributed computing.

outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves some or all of the unmatched rows of the tables that are being joined. See also *join.*

overloaded function. A function name for which multiple function instances exist.

P

package. An object containing a set of SQL statements that have been statically bound and that is available for processing. A package is sometimes also called an *application package*.

package list. An ordered list of package names that may be used to extend an application plan.

package name. The name of an object that is created by a BIND PACKAGE or REBIND PACKAGE command. The object is a bound version of a database request module (DBRM). The name consists of a location name, a collection ID, a package ID, and a version ID.

page. A unit of storage within a table space (4 KB, 8 KB, 16 KB, or 32 KB) or index space (4 KB). In a table space, a page contains one or more rows of a table. In a LOB table space, a LOB value can span more than one page, but no more than one LOB value is stored on a page.

page set. Another way to refer to a table space or index space. Each page set consists of a collection of VSAM data sets.

page set recovery pending (PSRCP). A restrictive state of an index space. In this case, the entire page set must be recovered. Recovery of a logical part is prohibited.

panel. A predefined display image that defines the locations and characteristics of display fields on a display surface (for example, a *menu panel*).

parallel complex. A cluster of machines that work together to handle multiple transactions and applications.

parallel group. A set of consecutive operations that execute in parallel and that have the same number of parallel tasks.

parallel I/O processing. A form of I/O processing in which DB2 initiates multiple concurrent requests for a single user query and performs I/O processing concurrently (in *parallel*) on multiple data partitions.

parallelism assistant. In Sysplex query parallelism, a DB2 subsystem that helps to process parts of a parallel query that originates on another DB2 subsystem in the data sharing group.

parallelism coordinator. In Sysplex query parallelism, the DB2 subsystem from which the parallel query originates.

Parallel Sysplex. A set of z/OS systems that communicate and cooperate with each other through certain multisystem hardware components and software services to process customer workloads.

parallel task. The execution unit that is dynamically created to process a query in parallel. A parallel task is implemented by a z/OS service request block.

parameter marker. A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a host variable could appear if the statement string were a static SQL statement.

parameter-name. An SQL identifier that designates a parameter in an SQL procedure or an SQL function.

parent key. A primary key or unique key in the parent table of a referential constraint. The values of a parent key determine the valid values of the foreign key in the referential constraint.

| **parent lock.** For explicit hierarchical locking, a lock
| that is held on a resource that might have child locks
| that are lower in the hierarchy. A parent lock is usually
| the table space lock or the partition intent lock. See also
| *child lock*.

parent row. A row whose primary key value is the foreign key value of a dependent row.

parent table. A table whose primary key is referenced by the foreign key of a dependent table.

parent table space. A table space that contains a parent table. A table space containing a dependent of that table is a dependent table space.

participant. An entity other than the commit coordinator that takes part in the commit process. The term participant is synonymous with *agent* in SNA.

partition. A portion of a page set. Each partition corresponds to a single, independently extendable data set. Partitions can be extended to a maximum size of 1, 2, or 4 GB, depending on the number of partitions in the partitioned page set. All partitions of a given page set have the same maximum size.

partitioned data set (PDS). A data set in disk storage that is divided into partitions, which are called members. Each partition can contain a program, part of a program, or data. The term partitioned data set is synonymous with program library.

| **partitioned index.** An index that is physically
| partitioned. Both partitioning indexes and secondary
| indexes can be partitioned.

partitioned page set. A partitioned table space or an index space. Header pages, space map pages, data pages, and index pages reference data only within the scope of the partition.

partitioned table space. A table space that is subdivided into parts (based on index key range), each of which can be processed independently by utilities.

partitioning index. An index in which the leftmost columns are the partitioning columns of the table. The index can be partitioned or nonpartitioned.

| **partition pruning.** The removal from consideration of inapplicable partitions through setting up predicates in a query on a partitioned table to access only certain partitions to satisfy the query.

partner logical unit. An access point in the SNA network that is connected to the local DB2 subsystem by way of a VTAM conversation.

path. See *SQL path*.

PCT. Program control table (in CICS).

PDS. Partitioned data set.

piece. A data set of a nonpartitioned page set.

physical claim. A claim on an entire nonpartitioning index.

physical consistency. The state of a page that is not in a partially changed state.

physical drain. A drain on an entire nonpartitioning index.

physical lock (P-lock). A type of lock that DB2 acquires to provide consistency of data that is cached in different DB2 subsystems. Physical locks are used only in data sharing environments. Contrast with *logical lock (L-lock)*.

physical lock contention. Conflicting states of the requesters for a physical lock. See also *negotiable lock*.

physically complete. The state in which the concurrent copy process is completed and the output data set has been created.

plan. See *application plan*.

plan allocation. The process of allocating DB2 resources to a plan in preparation for execution.

plan member. The bound copy of a DBRM that is identified in the member clause.

plan name. The name of an application plan.

plan segmentation. The dividing of each plan into sections. When a section is needed, it is independently brought into the EDM pool.

P-lock. Physical lock.

PLT. Program list table (in CICS).

point of consistency. A time when all recoverable data that an application accesses is consistent with other data. The term point of consistency is synonymous with *sync point* or *commit point*.

policy. See *CFRM policy*.

Portable Operating System Interface (POSIX). The IEEE operating system interface standard, which defines the Pthread standard of threading. See also *Pthread*.

POSIX. Portable Operating System Interface.

postponed abort UR. A unit of recovery that was inflight or in-abort, was interrupted by system failure or cancellation, and did not complete backout during restart.

PPT. (1) Processing program table (in CICS). (2) Program properties table (in z/OS).

precision. In SQL, the total number of digits in a decimal number (called the *size* in the C language). In the C language, the number of digits to the right of the decimal point (called the *scale* in SQL). The DB2 library uses the SQL terms.

precompilation. A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.

predicate. An element of a search condition that expresses or implies a comparison operation.

prefix. A code at the beginning of a message or record.

preformat. The process of preparing a VSAM ESDS for DB2 use, by writing specific data patterns.

prepare. The first phase of a two-phase commit process in which all participants are requested to prepare for commit.

prepared SQL statement. A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.

presumed-abort. An optimization of the presumed-nothing two-phase commit protocol that reduces the number of recovery log records, the duration of state maintenance, and the number of messages between coordinator and participant. The optimization also modifies the indoubt resolution responsibility.

presumed-nothing. The standard two-phase commit protocol that defines coordinator and participant responsibilities, relative to logical unit of work states, recovery logging, and indoubt resolution.

primary authorization ID. The authorization ID that is used to identify the application process to DB2.

primary group buffer pool. For a duplexed group buffer pool, the structure that is used to maintain the coherency of cached data. This structure is used for page registration and cross-invalidation. The z/OS equivalent is *old* structure. Compare with *secondary group buffer pool*.

primary index. An index that enforces the uniqueness of a primary key.

primary key. In a relational database, a unique, nonnull key that is part of the definition of a table. A table cannot be defined as a parent unless it has a unique key or primary key.

principal. An entity that can communicate securely with another entity. In Kerberos, principals are represented as entries in the Kerberos registry database and include users, servers, computers, and others.

principal name. The name by which a principal is known to the DCE security services.

private connection. A communications connection that is specific to DB2.

private protocol access. A method of accessing distributed data by which you can direct a query to another DB2 system. Contrast with *DRDA access*.

private protocol connection. A DB2 private connection of the application process. See also *private connection*.

privilege. The capability of performing a specific function, sometimes on a specific object. The types of privileges are:

> **explicit privileges**, which have names and are held as the result of SQL GRANT and REVOKE statements. For example, the SELECT privilege.
> **implicit privileges**, which accompany the ownership of an object, such as the privilege to drop a synonym that one owns, or the holding of an authority, such as the privilege of SYSADM authority to terminate any utility job.

privilege set. For the installation SYSADM ID, the set of all possible privileges. For any other authorization ID, the set of all privileges that are recorded for that ID in the DB2 catalog.

process. In DB2, the unit to which DB2 allocates resources and locks. Sometimes called an *application process*, a process involves the execution of one or more programs. The execution of an SQL statement is always associated with some process. The means of initiating and terminating a process are dependent on the environment.

program. A single, compilable collection of executable statements in a programming language.

program temporary fix (PTF). A solution or bypass of a problem that is diagnosed as a result of a defect in a current unaltered release of a licensed program. An authorized program analysis report (APAR) fix is corrective service for an existing problem. A PTF is preventive service for problems that might be encountered by other users of the product. A PTF is *temporary*, because a permanent fix is usually not incorporated into the product until its next release.

protected conversation. A VTAM conversation that supports two-phase commit flows.

PSRCP. Page set recovery pending.

PTF. Program temporary fix.

Pthread. The POSIX threading standard model for splitting an application into subtasks. The Pthread standard includes functions for creating threads, terminating threads, synchronizing threads through locking, and other thread control facilities.

Q

QMF™. Query Management Facility.

QSAM. Queued sequential access method.

query. A component of certain SQL statements that specifies a result table.

query block. The part of a query that is represented by one of the FROM clauses. Each FROM clause can have multiple query blocks, depending on DB2's internal processing of the query.

query CP parallelism. Parallel execution of a single query, which is accomplished by using multiple tasks. See also *Sysplex query parallelism*.

query I/O parallelism. Parallel access of data, which is accomplished by triggering multiple I/O requests within a single query.

queued sequential access method (QSAM). An extended version of the basic sequential access method (BSAM). When this method is used, a queue of data blocks is formed. Input data blocks await processing, and output data blocks await transfer to auxiliary storage or to an output device.

quiesce point. A point at which data is consistent as a result of running the DB2 QUIESCE utility.

quiesced member state. A state of a member of a data sharing group. An active member becomes quiesced when a STOP DB2 command takes effect without a failure. If the member's task, address space, or z/OS system fails before the command takes effect, the member state is failed.

R

| **RACF.** Resource Access Control Facility, which is a
| component of the z/OS Security Server.

RAMAC®. IBM family of enterprise disk storage system products.

RBA. Relative byte address.

RCT. Resource control table (in CICS attachment facility).

RDB. Relational database.

RDBMS. Relational database management system.

RDBNAM. Relational database name.

RDF. Record definition field.

read stability (RS). An isolation level that is similar to repeatable read but does not completely isolate an application process from all other concurrently executing application processes. Under level RS, an application that issues the same query more than once might read additional rows that were inserted and committed by a concurrently executing application process.

rebind. The creation of a new application plan for an application program that has been bound previously. If, for example, you have added an index for a table that your application accesses, you must rebind the application in order to take advantage of that index.

rebuild. The process of reallocating a coupling facility structure. For the shared communications area (SCA) and lock structure, the structure is repopulated; for the group buffer pool, changed pages are usually cast out to disk, and the new structure is populated only with changed pages that were not successfully cast out.

RECFM. Record format.

record. The storage representation of a row or other data.

record identifier (RID). A unique identifier that DB2 uses internally to identify a row of data in a table. Compare with *row ID*.

record identifier (RID) pool. An area of main storage that is used for sorting record identifiers during list-prefetch processing.

record length. The sum of the length of all the columns in a table, which is the length of the data as it is physically stored in the database. Records can be fixed length or varying length, depending on how the columns are defined. If all columns are fixed-length columns, the record is a fixed-length record. If one or more columns are varying-length columns, the record is a varying-length column.

Recoverable Resource Manager Services attachment facility (RRSAF). A DB2 subcomponent that uses Resource Recovery Services to coordinate resource commitment between DB2 and all other resource managers that also use RRS in a z/OS system.

recovery. The process of rebuilding databases after a system failure.

recovery log. A collection of records that describes the events that occur during DB2 execution and indicates their sequence. The recorded information is used for recovery in the event of a failure during DB2 execution.

recovery manager. (1) A subcomponent that supplies coordination services that control the interaction of DB2 resource managers during commit, abort, checkpoint, and restart processes. The recovery manager also supports the recovery mechanisms of other subsystems (for example, IMS) by acting as a participant in the other subsystem's process for protecting data that has reached a point of consistency. (2) A coordinator or a participant (or both), in the execution of a two-phase commit, that can access a recovery log that maintains the state of the logical unit of work and names the immediate upstream coordinator and downstream participants.

recovery pending (RECP). A condition that prevents SQL access to a table space that needs to be recovered.

recovery token. An identifier for an element that is used in recovery (for example, NID or URID).

RECP. Recovery pending.

redo. A state of a unit of recovery that indicates that changes are to be reapplied to the disk media to ensure data integrity.

reentrant. Executable code that can reside in storage as one shared copy for all threads. Reentrant code is not self-modifying and provides separate storage areas for each thread. Reentrancy is a compiler and operating system concept, and reentrancy alone is not enough to guarantee logically consistent results when multithreading. See also *threadsafe*.

referential constraint. The requirement that nonnull values of a designated foreign key are valid only if they equal values of the primary key of a designated table.

referential integrity. The state of a database in which all values of all foreign keys are valid. Maintaining referential integrity requires the enforcement of referential constraints on all operations that change the data in a table on which the referential constraints are defined.

referential structure. A set of tables and relationships that includes at least one table and, for every table in the set, all the relationships in which that table participates and all the tables to which it is related.

| **refresh age.** The time duration between the current
| time and the time during which a materialized query
| table was last refreshed.

registry. See *registry database*.

registry database. A database of security information about principals, groups, organizations, accounts, and security policies.

relational database (RDB). A database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

relational database management system (RDBMS). A collection of hardware and software that organizes and provides access to a relational database.

relational database name (RDBNAM). A unique identifier for an RDBMS within a network. In DB2, this must be the value in the LOCATION column of table SYSIBM.LOCATIONS in the CDB. DB2 publications refer to the name of another RDBMS as a LOCATION value or a location name.

relationship. A defined connection between the rows of a table or the rows of two tables. A relationship is the internal representation of a referential constraint.

relative byte address (RBA). The offset of a data record or control interval from the beginning of the storage space that is allocated to the data set or file to which it belongs.

remigration. The process of returning to a current release of DB2 following a fallback to a previous release. This procedure constitutes another migration process.

remote. Any object that is maintained by a remote DB2 subsystem (that is, by a DB2 subsystem other than the local one). A *remote view*, for example, is a view that is maintained by a remote DB2 subsystem. Contrast with *local.*

remote attach request. A request by a remote location to attach to the local DB2 subsystem. Specifically, the request that is sent is an SNA Function Management Header 5.

remote subsystem. Any relational DBMS, except the *local subsystem*, with which the user or application can communicate. The subsystem need not be remote in any physical sense, and might even operate on the same processor under the same z/OS system.

reoptimization. The DB2 process of reconsidering the access path of an SQL statement at run time; during reoptimization, DB2 uses the values of host variables, parameter markers, or special registers.

REORG pending (REORP). A condition that restricts SQL access and most utility access to an object that must be reorganized.

REORP. REORG pending.

repeatable read (RR). The isolation level that provides maximum protection from other executing application programs. When an application program executes with repeatable read protection, rows that the program references cannot be changed by other programs until the program reaches a commit point.

repeating group. A situation in which an entity includes multiple attributes that are inherently the same. The presence of a repeating group violates the requirement of first normal form. In an entity that satisfies the requirement of first normal form, each attribute is independent and unique in its meaning and its name. See also *normalization*.

replay detection mechanism. A method that allows a principal to detect whether a request is a valid request from a source that can be trusted or whether an untrustworthy entity has captured information from a previous exchange and is replaying the information exchange to gain access to the principal.

request commit. The vote that is submitted to the prepare phase if the participant has modified data and is prepared to commit or roll back.

requester. The source of a request to access data at a remote server. In the DB2 environment, the requester function is provided by the distributed data facility.

resource. The object of a lock or claim, which could be a table space, an index space, a data partition, an index partition, or a logical partition.

resource allocation. The part of plan allocation that deals specifically with the database resources.

resource control table (RCT). A construct of the CICS attachment facility, created by site-provided macro parameters, that defines authorization and access attributes for transactions or transaction groups.

resource definition online. A CICS feature that you use to define CICS resources online without assembling tables.

resource limit facility (RLF). A portion of DB2 code that prevents dynamic manipulative SQL statements from exceeding specified time limits. The resource limit facility is sometimes called the governor.

resource limit specification table (RLST). A site-defined table that specifies the limits to be enforced by the resource limit facility.

resource manager. (1) A function that is responsible for managing a particular resource and that guarantees the consistency of all updates made to recoverable resources within a logical unit of work. The resource that is being managed can be physical (for example, disk or main storage) or logical (for example, a particular type of system service). (2) A participant, in the execution of a two-phase commit, that has recoverable resources that could have been modified. The resource manager has access to a recovery log so that it can commit or roll back the effects of the logical unit of work to the recoverable resources.

restart pending (RESTP). A restrictive state of a page set or partition that indicates that restart (backout) work needs to be performed on the object. All access to the page set or partition is denied except for access by the:
- RECOVER POSTPONED command
- Automatic online backout (which DB2 invokes after restart if the system parameter LBACKOUT=AUTO)

RESTP. Restart pending.

result set. The set of rows that a stored procedure returns to a client application.

result set locator. A 4-byte value that DB2 uses to uniquely identify a query result set that a stored procedure returns.

result table. The set of rows that are specified by a SELECT statement.

retained lock. A MODIFY lock that a DB2 subsystem was holding at the time of a subsystem failure. The lock is retained in the coupling facility lock structure across a DB2 failure.

RID. Record identifier.

RID pool. Record identifier pool.

right outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of the second join operand. See also *join*.

RLF. Resource limit facility.

RLST. Resource limit specification table.

RMID. Resource manager identifier.

RO. Read-only access.

rollback. The process of restoring data that was changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with *commit*.

root page. The index page that is at the highest level (or the beginning point) in an index.

routine. A term that refers to either a user-defined function or a stored procedure.

row. The horizontal component of a table. A row consists of a sequence of values, one for each column of the table.

ROWID. Row identifier.

row identifier (ROWID). A value that uniquely identifies a row. This value is stored with the row and never changes.

row lock. A lock on a single row of data.

| **rowset.** A set of rows for which a cursor position is
| established.

rowset cursor. A cursor that is defined so that one or more rows can be returned as a rowset for a single FETCH statement, and the cursor is positioned on the set of rows that is fetched.

rowset-positioned access. The ability to retrieve multiple rows from a single FETCH statement.

| **row-positioned access.** The ability to retrieve a single
| row from a single FETCH statement.

row trigger. A trigger that is defined with the trigger granularity FOR EACH ROW.

RRE. Residual recovery entry (in IMS).

RRSAF. Recoverable Resource Manager Services attachment facility.

RS. Read stability.

RTT. Resource translation table.

RURE. Restart URE.

S

savepoint. A named entity that represents the state of data and schemas at a particular point in time within a unit of work. SQL statements exist to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents. The restoration of data and schemas to a savepoint is usually referred to as *rolling back to a savepoint*.

SBCS. Single-byte character set.

SCA. Shared communications area.

scalar function. An SQL operation that produces a single value from another value and is expressed as a function name, followed by a list of arguments that are enclosed in parentheses. Contrast with *column function*.

scale. In SQL, the number of digits to the right of the decimal point (called the *precision* in the C language). The DB2 library uses the SQL definition.

schema. (1) The organization or structure of a database. (2) A logical grouping for user-defined functions, distinct types, triggers, and stored procedures. When an object of one of these types is created, it is assigned to one schema, which is determined by the name of the object. For example, the following statement creates a distinct type T in schema C:

```
CREATE DISTINCT TYPE C.T ...
```

scrollability. The ability to use a cursor to fetch in either a forward or backward direction. The FETCH statement supports multiple fetch orientations to indicate the new position of the cursor. See also *fetch orientation*.

scrollable cursor. A cursor that can be moved in both a forward and a backward direction.

SDWA. System diagnostic work area.

search condition. A criterion for selecting rows from a table. A search condition consists of one or more predicates.

secondary authorization ID. An authorization ID that has been associated with a primary authorization ID by an authorization exit routine.

secondary group buffer pool. For a duplexed group buffer pool, the structure that is used to back up changed pages that are written to the primary group buffer pool. No page registration or cross-invalidation occurs using the secondary group buffer pool. The z/OS equivalent is *new* structure.

secondary index. A nonpartitioning index on a partitioned table.

section. The segment of a plan or package that contains the executable structures for a single SQL statement. For most SQL statements, one section in the plan exists for each SQL statement in the source program. However, for cursor-related statements, the DECLARE, OPEN, FETCH, and CLOSE statements reference the same section because they each refer to the SELECT statement that is named in the DECLARE CURSOR statement. SQL statements such as COMMIT, ROLLBACK, and some SET statements do not use a section.

segment. A group of pages that holds rows of a single table. See also *segmented table space*.

segmented table space. A table space that is divided into equal-sized groups of pages called segments. Segments are assigned to tables so that rows of different tables are never stored in the same segment.

self-referencing constraint. A referential constraint that defines a relationship in which a table is a dependent of itself.

self-referencing table. A table with a self-referencing constraint.

sensitive cursor. A cursor that is sensitive to changes that are made to the database after the result table has been materialized.

sequence. A user-defined object that generates a sequence of numeric values according to user specifications.

sequential data set. A non-DB2 data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. Several of the DB2 database utilities require sequential data sets.

sequential prefetch. A mechanism that triggers consecutive asynchronous I/O operations. Pages are fetched before they are required, and several pages are read with a single I/O operation.

serial cursor. A cursor that can be moved only in a forward direction.

serialized profile. A Java object that contains SQL statements and descriptions of host variables. The SQLJ translator produces a serialized profile for each connection context.

server. The target of a request from a remote requester. In the DB2 environment, the server function is provided by the distributed data facility, which is used to access DB2 data from remote applications.

server-side programming. A method for adding DB2 data into dynamic Web pages.

service class. An eight-character identifier that is used by the z/OS Workload Manager to associate user performance goals with a particular DDF thread or stored procedure. A service class is also used to classify work on parallelism assistants.

service request block. A unit of work that is scheduled to execute in another address space.

session. A link between two nodes in a VTAM network.

session protocols. The available set of SNA communication requests and responses.

shared communications area (SCA). A coupling facility list structure that a DB2 data sharing group uses for inter-DB2 communication.

share lock. A lock that prevents concurrently executing application processes from changing data, but not from reading data. Contrast with *exclusive lock*.

shift-in character. A special control character (X'0F') that is used in EBCDIC systems to denote that the subsequent bytes represent SBCS characters. See also *shift-out character*.

shift-out character. A special control character (X'0E') that is used in EBCDIC systems to denote that the subsequent bytes, up to the next shift-in control character, represent DBCS characters. See also *shift-in character*.

sign-on. A request that is made on behalf of an individual CICS or IMS application process by an attachment facility to enable DB2 to verify that it is authorized to use DB2 resources.

simple page set. A nonpartitioned page set. A simple page set initially consists of a single data set (page set piece). If and when that data set is extended to 2 GB, another data set is created, and so on, up to a total of 32 data sets. DB2 considers the data sets to be a single contiguous linear address space containing a maximum of 64 GB. Data is stored in the next available location within this address space without regard to any partitioning scheme.

simple table space. A table space that is neither partitioned nor segmented.

single-byte character set (SBCS). A set of characters in which each character is represented by a single byte. Contrast with *double-byte character set* or *multibyte character set*.

single-precision floating point number. A 32-bit approximate representation of a real number.

size. In the C language, the total number of digits in a decimal number (called the *precision* in SQL). The DB2 library uses the SQL term.

SMF. System Management Facilities.

SMP/E. System Modification Program/Extended.

SMS. Storage Management Subsystem.

SNA. Systems Network Architecture.

SNA network. The part of a network that conforms to the formats and protocols of Systems Network Architecture (SNA).

socket. A callable TCP/IP programming interface that TCP/IP network applications use to communicate with remote TCP/IP partners.

sourced function. A function that is implemented by another built-in or user-defined function that is already known to the database manager. This function can be a scalar function or a column (aggregating) function; it returns a single value from a set of values (for example, MAX or AVG). Contrast with *built-in function, external function,* and *SQL function*.

source program. A set of host language statements and SQL statements that is processed by an SQL precompiler.

| **source table.** A table that can be a base table, a view,
| a table expression, or a user-defined table function.

source type. An existing type that DB2 uses to internally represent a distinct type.

space. A sequence of one or more blank characters.

special register. A storage area that DB2 defines for an application process to use for storing information that can be referenced in SQL statements. Examples of special registers are USER and CURRENT DATE.

specific function name. A particular user-defined function that is known to the database manager by its specific name. Many specific user-defined functions can have the same function name. When a user-defined function is defined to the database, every function is assigned a specific name that is unique within its schema. Either the user can provide this name, or a default name is used.

SPUFI. SQL Processor Using File Input.

SQL. Structured Query Language.

SQL authorization ID (SQL ID). The authorization ID that is used for checking dynamic SQL statements in some situations.

SQLCA. SQL communication area.

SQL communication area (SQLCA). A structure that is used to provide an application program with information about the execution of its SQL statements.

SQL connection. An association between an application process and a local or remote application server or database server.

SQLDA. SQL descriptor area.

SQL descriptor area (SQLDA). A structure that describes input variables, output variables, or the columns of a result table.

SQL escape character. The symbol that is used to enclose an SQL delimited identifier. This symbol is the double quotation mark ("). See also *escape character*.

SQL function. A user-defined function in which the CREATE FUNCTION statement contains the source code. The source code is a single SQL expression that evaluates to a single value. The SQL user-defined function can return only one parameter.

SQL ID. SQL authorization ID.

SQLJ. Structured Query Language (SQL) that is embedded in the Java programming language.

SQL path. An ordered list of schema names that are used in the resolution of unqualified references to user-defined functions, distinct types, and stored procedures. In dynamic SQL, the current path is found in the CURRENT PATH special register. In static SQL, it is defined in the PATH bind option.

SQL procedure. A user-written program that can be invoked with the SQL CALL statement. Contrast with *external procedure*.

SQL processing conversation. Any conversation that requires access of DB2 data, either through an application or by dynamic query requests.

SQL Processor Using File Input (SPUFI). A facility of the TSO attachment subcomponent that enables the DB2I user to execute SQL statements without embedding them in an application program.

SQL return code. Either SQLCODE or SQLSTATE.

SQL routine. A user-defined function or stored procedure that is based on code that is written in SQL.

SQL statement coprocessor. An alternative to the DB2 precompiler that lets the user process SQL statements at compile time. The user invokes an SQL statement coprocessor by specifying a compiler option.

SQL string delimiter. A symbol that is used to enclose an SQL string constant. The SQL string delimiter is the apostrophe ('), except in COBOL applications, where the user assigns the symbol, which is either an apostrophe or a double quotation mark (").

SRB. Service request block.

SSI. Subsystem interface (in z/OS).

SSM. Subsystem member (in IMS).

stand-alone. An attribute of a program that means that it is capable of executing separately from DB2, without using DB2 services.

star join. A method of joining a dimension column of a fact table to the key column of the corresponding dimension table. See also *join*, *dimension*, and *star schema*.

star schema. The combination of a fact table (which contains most of the data) and a number of dimension tables. See also *star join*, *dimension*, and *dimension table*.

statement handle. In DB2 ODBC, the data object that contains information about an SQL statement that is managed by DB2 ODBC. This includes information such as dynamic arguments, bindings for dynamic arguments and columns, cursor information, result values, and status information. Each statement handle is associated with the connection handle.

statement string. For a dynamic SQL statement, the character string form of the statement.

statement trigger. A trigger that is defined with the trigger granularity FOR EACH STATEMENT.

| **static cursor.** A named control structure that does not
| change the size of the result table or the order of its
| rows after an application opens the cursor. Contrast with
| *dynamic cursor.*

static SQL. SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of host variables that are specified by the statement might change).

storage group. A named set of disks on which DB2 data can be stored.

stored procedure. A user-written application program that can be invoked through the use of the SQL CALL statement.

string. See *character string* or *graphic string*.

strong typing. A process that guarantees that only user-defined functions and operations that are defined on a distinct type can be applied to that type. For example, you cannot directly compare two currency types, such as Canadian dollars and U.S. dollars. But you can provide a user-defined function to convert one currency to the other and then do the comparison.

structure. (1) A name that refers collectively to different types of DB2 objects, such as tables, databases, views, indexes, and table spaces. (2) A construct that uses z/OS to map and manage storage on a coupling facility. See also *cache structure*, *list structure*, or *lock structure*.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

structure owner. In relation to group buffer pools, the DB2 member that is responsible for the following activities:

- Coordinating rebuild, checkpoint, and damage assessment processing
- Monitoring the group buffer pool threshold and notifying castout owners when the threshold has been reached

subcomponent. A group of closely related DB2 modules that work together to provide a general function.

subject table. The table for which a trigger is created. When the defined triggering event occurs on this table, the trigger is activated.

subpage. The unit into which a physical index page can be divided.

subquery. A SELECT statement within the WHERE or HAVING clause of another SQL statement; a nested SQL statement.

subselect. That form of a query that does not include an ORDER BY clause, an UPDATE clause, or UNION operators.

substitution character. A unique character that is substituted during character conversion for any characters in the source program that do not have a match in the target coding representation.

subsystem. A distinct instance of a relational database management system (RDBMS).

surrogate pair. A coded representation for a single character that consists of a sequence of two 16-bit code units, in which the first value of the pair is a high-surrogate code unit in the range U+D800 through U+DBFF, and the second value is a low-surrogate code unit in the range U+DC00 through U+DFFF. Surrogate pairs provide an extension mechanism for encoding 917 476 characters without requiring the use of 32-bit characters.

SVC dump. A dump that is issued when a z/OS or a DB2 functional recovery routine detects an error.

sync point. See *commit point*.

syncpoint tree. The tree of recovery managers and resource managers that are involved in a logical unit of work, starting with the recovery manager, that make the final commit decision.

synonym. In SQL, an alternative name for a table or view. Synonyms can be used to refer only to objects at the subsystem in which the synonym is defined.

syntactic character set. A set of 81 graphic characters that are registered in the IBM registry as character set 00640. This set was originally recommended to the programming language community to be used for syntactic purposes toward maximizing portability and interchangeability across systems and country boundaries. It is contained in most of the primary registered character sets, with a few exceptions. See also *invariant character set*.

Sysplex. See *Parallel Sysplex*.

Sysplex query parallelism. Parallel execution of a single query that is accomplished by using multiple tasks on more than one DB2 subsystem. See also *query CP parallelism*.

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

system agent. A work request that DB2 creates internally such as prefetch processing, deferred writes, and service tasks.

system conversation. The conversation that two DB2 subsystems must establish to process system messages before any distributed processing can begin.

system diagnostic work area (SDWA). The data that is recorded in a SYS1.LOGREC entry that describes a program or hardware error.

system-directed connection. A connection that a relational DBMS manages by processing SQL statements with three-part names.

System Modification Program/Extended (SMP/E). A z/OS tool for making software changes in programming systems (such as DB2) and for controlling those changes.

Systems Network Architecture (SNA). The description of the logical structure, formats, protocols, and operational sequences for transmitting information through and controlling the configuration and operation of networks.

SYS1.DUMPxx data set. A data set that contains a system dump (in z/OS).

SYS1.LOGREC. A service aid that contains important information about program and hardware errors (in z/OS).

T

table. A named data object consisting of a specific number of columns and some number of unordered rows. See also *base table* or *temporary table*.

table-controlled partitioning. A type of partitioning in which partition boundaries for a partitioned table are controlled by values that are defined in the CREATE TABLE statement. Partition limits are saved in the LIMITKEY_INTERNAL column of the SYSIBM.SYSTABLEPART catalog table.

table function. A function that receives a set of arguments and returns a table to the SQL statement that references the function. A table function can be referenced only in the FROM clause of a subselect.

table locator. A mechanism that allows access to trigger transition tables in the FROM clause of SELECT statements, in the subselect of INSERT statements, or from within user-defined functions. A table locator is a fullword integer value that represents a transition table.

table space. A page set that is used to store the records in one or more tables.

table space set. A set of table spaces and partitions that should be recovered together for one of these reasons:
- Each of them contains a table that is a parent or descendent of a table in one of the others.
- The set contains a base table and associated auxiliary tables.

A table space set can contain both types of relationships.

task control block (TCB). A z/OS control block that is used to communicate information about tasks within an address space that are connected to DB2. See also *address space connection*.

TB. Terabyte (1 099 511 627 776 bytes).

TCB. Task control block (in z/OS).

TCP/IP. A network communication protocol that computer systems use to exchange information across telecommunication links.

TCP/IP port. A 2-byte value that identifies an end user or a TCP/IP network application within a TCP/IP host.

template. A DB2 utilities output data set descriptor that is used for dynamic allocation. A template is defined by the TEMPLATE utility control statement.

temporary table. A table that holds temporary data. Temporary tables are useful for holding or sorting intermediate results from queries that contain a large number of rows. The two types of temporary table, which are created by different SQL statements, are the created temporary table and the declared temporary table. Contrast with *result table*. See also *created temporary table* and *declared temporary table*.

Terminal Monitor Program (TMP). A program that provides an interface between terminal users and command processors and has access to many system services (in z/OS).

thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services. Most DB2 functions execute under a thread structure. See also *allied thread* and *database access thread*.

threadsafe. A characteristic of code that allows multithreading both by providing private storage areas for each thread, and by properly serializing shared (global) storage areas.

three-part name. The full name of a table, view, or alias. It consists of a location name, authorization ID, and an object name, separated by a period.

time. A three-part value that designates a time of day in hours, minutes, and seconds.

time duration. A decimal integer that represents a number of hours, minutes, and seconds.

timeout. Abnormal termination of either the DB2 subsystem or of an application because of the unavailability of resources. Installation specifications are set to determine both the amount of time DB2 is to wait for IRLM services after starting, and the amount of time IRLM is to wait if a resource that an application requests is unavailable. If either of these time specifications is exceeded, a timeout is declared.

Time-Sharing Option (TSO). An option in MVS that provides interactive time sharing from remote terminals.

timestamp. A seven-part value that consists of a date and time. The timestamp is expressed in years, months, days, hours, minutes, seconds, and microseconds.

TMP. Terminal Monitor Program.

to-do. A state of a unit of recovery that indicates that the unit of recovery's changes to recoverable DB2 resources are indoubt and must either be applied to the disk media or backed out, as determined by the commit coordinator.

trace. A DB2 facility that provides the ability to monitor and collect DB2 monitoring, auditing, performance, accounting, statistics, and serviceability (global) data.

transaction lock. A lock that is used to control concurrent execution of SQL statements.

transaction program name. In SNA LU 6.2 conversations, the name of the program at the remote logical unit that is to be the other half of the conversation.

| **transient XML data type.** A data type for XML values
| that exists only during query processing.

transition table. A temporary table that contains all the affected rows of the subject table in their state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the table of changed rows in the old state or the new state.

transition variable. A variable that contains a column value of the affected row of the subject table in its state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the set of old values or the set of new values.

tree structure. A data structure that represents entities in nodes, with a most one parent node for each node, and with only one root node.

trigger. A set of SQL statements that are stored in a DB2 database and executed when a certain event occurs in a DB2 table.

trigger activation. The process that occurs when the trigger event that is defined in a trigger definition is executed. Trigger activation consists of the evaluation of the triggered action condition and conditional execution of the triggered SQL statements.

trigger activation time. An indication in the trigger definition of whether the trigger should be activated before or after the triggered event.

trigger body. The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. A trigger body is also called *triggered SQL statements*.

trigger cascading. The process that occurs when the triggered action of a trigger causes the activation of another trigger.

triggered action. The SQL logic that is performed when a trigger is activated. The triggered action consists of an optional triggered action condition and a set of triggered SQL statements that are executed only if the condition evaluates to true.

triggered action condition. An optional part of the triggered action. This Boolean condition appears as a WHEN clause and specifies a condition that DB2 evaluates to determine if the triggered SQL statements should be executed.

triggered SQL statements. The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. Triggered SQL statements are also called the *trigger body*.

trigger granularity. A characteristic of a trigger, which determines whether the trigger is activated:
- Only once for the triggering SQL statement
- Once for each row that the SQL statement modifies

triggering event. The specified operation in a trigger definition that causes the activation of that trigger. The triggering event is comprised of a triggering operation (INSERT, UPDATE, or DELETE) and a subject table on which the operation is performed.

triggering SQL operation. The SQL operation that causes a trigger to be activated when performed on the subject table.

trigger package. A package that is created when a CREATE TRIGGER statement is executed. The package is executed when the trigger is activated.

TSO. Time-Sharing Option.

TSO attachment facility. A DB2 facility consisting of the DSN command processor and DB2I. Applications that are not written for the CICS or IMS environments can run under the TSO attachment facility.

typed parameter marker. A parameter marker that is specified along with its target data type. It has the general form:

```
CAST(? AS data-type)
```

type 1 indexes. Indexes that were created by a release of DB2 before DB2 Version 4 or that are specified as type 1 indexes in Version 4. Contrast with *type 2 indexes*. As of Version 8, type 1 indexes are no longer supported.

type 2 indexes. Indexes that are created on a release of DB2 after Version 7 or that are specified as type 2 indexes in Version 4 or later.

U

UCS-2. Universal Character Set, coded in 2 octets, which means that characters are represented in 16-bits per character.

UDF. User-defined function.

UDT. User-defined data type. In DB2 UDB for z/OS, the term *distinct type* is used instead of user-defined data type. See *distinct type*.

uncommitted read (UR). The isolation level that allows an application to read uncommitted data.

underlying view. The view on which another view is directly or indirectly defined.

undo. A state of a unit of recovery that indicates that the changes that the unit of recovery made to recoverable DB2 resources must be backed out.

Unicode. A standard that parallels the ISO-10646 standard. Several implementations of the Unicode standard exist, all of which have the ability to represent a large percentage of the characters that are contained in the many scripts that are used throughout the world.

uniform resource locator (URL). A Web address, which offers a way of naming and locating specific items on the Web.

union. An SQL operation that combines the results of two SELECT statements. Unions are often used to merge lists of values that are obtained from several tables.

unique constraint. An SQL rule that no two values in a primary key, or in the key of a unique index, can be the same.

unique index. An index that ensures that no identical key values are stored in a column or a set of columns in a table.

unit of recovery. A recoverable sequence of operations within a single resource manager, such as an instance of DB2. Contrast with *unit of work*.

unit of recovery identifier (URID). The LOGRBA of the first log record for a unit of recovery. The URID also appears in all subsequent log records for that unit of recovery.

unit of work. A recoverable sequence of operations within an application process. At any time, an application process is a single unit of work, but the life of an application process can involve many units of work as a result of commit or rollback operations. In a *multisite update* operation, a single unit of work can include several *units of recovery*. Contrast with *unit of recovery*.

Universal Unique Identifier (UUID). An identifier that is immutable and unique across time and space (in z/OS).

unlock. The act of releasing an object or system resource that was previously locked and returning it to general availability within DB2.

untyped parameter marker. A parameter marker that is specified without its target data type. It has the form of a single question mark (?).

updatability. The ability of a cursor to perform positioned updates and deletes. The updatability of a cursor can be influenced by the SELECT statement and the cursor sensitivity option that is specified on the DECLARE CURSOR statement.

update hole. The location on which a cursor is positioned when a row in a result table is fetched again and the new values no longer satisfy the search condition. DB2 marks a row in the result table as an update hole when an update to the corresponding row in the database causes that row to no longer qualify for the result table.

update trigger. A trigger that is defined with the triggering SQL operation UPDATE.

upstream. The node in the syncpoint tree that is responsible, in addition to other recovery or resource managers, for coordinating the execution of a two-phase commit.

UR. Uncommitted read.

URE. Unit of recovery element.

URID . Unit of recovery identifier.

URL. Uniform resource locator.

user-defined data type (UDT). See *distinct type*.

user-defined function (UDF). A function that is defined to DB2 by using the CREATE FUNCTION statement and that can be referenced thereafter in SQL statements. A user-defined function can be an *external function,* a *sourced function,* or an *SQL function.* Contrast with *built-in function*.

user view. In logical data modeling, a model or representation of critical information that the business requires.

UTF-8. Unicode Transformation Format, 8-bit encoding form, which is designed for ease of use with existing ASCII-based systems. The CCSID value for data in UTF-8 format is 1208. DB2 UDB for z/OS supports UTF-8 in mixed data fields.

UTF-16. Unicode Transformation Format, 16-bit encoding form, which is designed to provide code values for over a million characters and a superset of UCS-2. The CCSID value for data in UTF-16 format is 1200. DB2 UDB for z/OS supports UTF-16 in graphic data fields.

UUID. Universal Unique Identifier.

V

value. The smallest unit of data that is manipulated in SQL.

variable. A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a variable. Contrast with *constant*.

variant function. See *nondeterministic function*.

varying-length string. A character or graphic string whose length varies within set limits. Contrast with *fixed-length string*.

version. A member of a set of similar programs, DBRMs, packages, or LOBs.

> **A version of a program** is the source code that is produced by precompiling the program. The program version is identified by the program name and a timestamp (consistency token).
> **A version of a DBRM** is the DBRM that is produced by precompiling a program. The DBRM version is identified by the same program name and timestamp as a corresponding program version.
> **A version of a package** is the result of binding a DBRM within a particular database system. The package version is identified by the same program name and consistency token as the DBRM.
> **A version of a LOB** is a copy of a LOB value at a point in time. The version number for a LOB is stored in the auxiliary index entry for the LOB.

view. An alternative representation of data from one or more tables. A view can include all or some of the columns that are contained in tables on which it is defined.

view check option. An option that specifies whether every row that is inserted or updated through a view must conform to the definition of that view. A view check option can be specified with the WITH CASCADED

CHECK OPTION, WITH CHECK OPTION, or WITH LOCAL CHECK OPTION clauses of the CREATE VIEW statement.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed- and varying-length records on disk devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number (in z/OS).

Virtual Telecommunications Access Method (VTAM). An IBM licensed program that controls communication and the flow of data in an SNA network (in z/OS).

| **volatile table.** A table for which SQL operations
| choose index access whenever possible.

VSAM. Virtual Storage Access Method.

VTAM. Virtual Telecommunication Access Method (in z/OS).

W

warm start. The normal DB2 restart process, which involves reading and processing log records so that data that is under the control of DB2 is consistent. Contrast with *cold start*.

WLM application environment. A z/OS Workload Manager attribute that is associated with one or more stored procedures. The WLM application environment determines the address space in which a given DB2 stored procedure runs.

write to operator (WTO). An optional user-coded service that allows a message to be written to the system console operator informing the operator of errors and unusual system conditions that might need to be corrected (in z/OS).

WTO. Write to operator.

WTOR. Write to operator (WTO) with reply.

X

XCF. See *cross-system coupling facility*.

XES. See *cross-system extended services*.

| **XML.** See *Extensible Markup Language*.

| **XML attribute.** A name-value pair within a tagged XML
| element that modifies certain features of the element.

XML element. A logical structure in an XML document that is delimited by a start and an end tag.

XML node. The smallest unit of valid, complete structure in a document. For example, a node can represent an element, an attribute, or a text string.

| **XML publishing functions.** Functions that return XML
| values from SQL values.

X/Open. An independent, worldwide open systems organization that is supported by most of the world's largest information systems suppliers, user organizations, and software companies. X/Open's goal is to increase the portability of applications by combining existing and emerging standards.

XRF. Extended recovery facility.

Z

| **z/OS.** An operating system for the eServer™ product
| line that supports 64-bit real and virtual storage.

z/OS Distributed Computing Environment (z/OS DCE). A set of technologies that are provided by the Open Software Foundation to implement distributed computing.

Bibliography

DB2 Universal Database for z/OS Version 8 product information:

The following information about Version 8 of DB2 UDB for z/OS is available in both printed and softcopy formats:

- *DB2 Administration Guide*, SC18-7413
- *DB2 Application Programming and SQL Guide*, SC18-7415
- *DB2 Application Programming Guide and Reference for Java*, SC18-7414
- *DB2 Command Reference*, SC18-7416
- *DB2 Data Sharing: Planning and Administration*, SC18-7417
- *DB2 Diagnosis Guide and Reference*, LY37-3201
- *DB2 Diagnostic Quick Reference Card*, LY37-3202
- *DB2 Installation Guide*, GC18-7418
- *DB2 Licensed Program Specifications*, GC18-7420
- *DB2 Messages and Codes*, GC18-7422
- *DB2 ODBC Guide and Reference*, SC18-7423
- *DB2 Reference Summary*, SX26-3853
- *DB2 Release Planning Guide*, SC18-7425
- *DB2 SQL Reference*, SC18-7426
- *DB2 Utility Guide and Reference*, SC18-7427
- *DB2 What's New?*, GC18-7428
- *DB2 XML Extender for z/OS Administration and Programming*, SC18-7431
- *Program Directory for IBM DB2 Universal Database for z/OS*, GI10-8566

The following information is provided in softcopy format only:

- *DB2 Image, Audio, and Video Extenders Administration and Programming* (Version 7 level)
- *DB2 Net Search Extender Administration and Programming Guide* (Version 7 level)
- *DB2 RACF Access Control Module Guide* (Version 8 level)
- *DB2 Reference for Remote DRDA Requesters and Servers* (Version 8 level)
- *DB2 Text Extender Administration and Programming* (Version 7 level)

You can find DB2 UDB for z/OS information on the library Web page at www.ibm.com/db2/zos/v8books.html

The preceding information is published by IBM. One additional book, which is written by IBM and published by Pearson Education, Inc., is *The Official Introduction to DB2 UDB for z/OS*, ISBN 0-13-147750-1. This book provides an overview of the Version 8 DB2 UDB for z/OS product and is recommended reading for people who are preparing to take Certification Exam 700: DB2 UDB V8.1 Family Fundamentals.

Books and resources about related products:

APL2®

- *APL2 Programming Guide*, SH21-1072
- *APL2 Programming: Language Reference*, SH21-1061
- *APL2 Programming: Using Structured Query Language (SQL)*, SH21-1057

BookManager® READ/MVS

- *BookManager READ/MVS V1R3: Installation Planning & Customization*, SC38-2035

C language: IBM C/C++ for z/OS

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821

Character Data Representation Architecture

- *Character Data Representation Architecture Overview*, GC09-2207
- *Character Data Representation Architecture Reference and Registry*, SC09-2190

CICS Transaction Server for z/OS

The publication order numbers below are for Version 2 Release 2 and Version 2 Release 3 (with the release 2 number listed first).

- *CICS Transaction Server for z/OS Information Center*, SK3T-6903 or SK3T-6957.
- *CICS Transaction Server for z/OS Application Programming Guide*, SC34-5993 or SC34-6231
- *CICS Transaction Server for z/OS Application Programming Reference*, SC34-5994 or SC34-6232
- *CICS Transaction Server for z/OS CICS-RACF Security Guide*, SC34-6011 or SC34-6249

- *CICS Transaction Server for z/OS CICS Supplied Transactions*, SC34-5992 or SC34-6230
- *CICS Transaction Server for z/OS Customization Guide*, SC34-5989 or SC34-6227
- *CICS Transaction Server for z/OS Data Areas*, LY33-6100 or LY33-6103
- *CICS Transaction Server for z/OS DB2 Guide*, SC34-6014 or SC34-6252
- *CICS Transaction Server for z/OS External Interfaces Guide*, SC34-6006 or SC34-6244
- *CICS Transaction Server for z/OS Installation Guide*, GC34-5985 or GC34-6224
- *CICS Transaction Server for z/OS Intercommunication Guide*, SC34-6005 or SC34-6243
- *CICS Transaction Server for z/OS Messages and Codes*, GC34-6003 or GC34-6241
- *CICS Transaction Server for z/OS Operations and Utilities Guide*, SC34-5991 or SC34-6229
- *CICS Transaction Server for z/OS Performance Guide*, SC34-6009 or SC34-6247
- *CICS Transaction Server for z/OS Problem Determination Guide*, SC34-6002 or SC34-6239
- *CICS Transaction Server for z/OS Release Guide*, GC34-5983 or GC34-6218
- *CICS Transaction Server for z/OS Resource Definition Guide*, SC34-5990 or SC34-6228
- *CICS Transaction Server for z/OS System Definition Guide*, SC34-5988 or SC34–6226
- *CICS Transaction Server for z/OS System Programming Reference*, SC34-5595 or SC34–6233

CICS Transaction Server for OS/390
- *CICS Transaction Server for OS/390 Application Programming Guide*, SC33-1687
- *CICS Transaction Server for OS/390 DB2 Guide*, SC33-1939
- *CICS Transaction Server for OS/390 External Interfaces Guide*, SC33-1944
- *CICS Transaction Server for OS/390 Resource Definition Guide*, SC33-1684

COBOL: IBM COBOL
- *IBM COBOL Language Reference*, SC27-1408
- *IBM COBOL for MVS & VM Programming Guide*, SC27-1412

Database Design
- *DB2 for z/OS and OS/390 Development for Performance Volume I* by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-605-2
- *DB2 for z/OS and OS/390 Development for Performance Volume II* by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-606-0
- *Handbook of Relational Database Design* by C. Fleming and B. Von Halle, Addison Wesley, ISBN 0-20111-434-8

DB2 Administration Tool
- *DB2 Administration Tool for z/OS User's Guide and Reference*, available on the Web at www.ibm.com/software/data/db2imstools/library.html

DB2 Buffer Pool Analyzer for z/OS
- *DB2 Buffer Pool Tool for z/OS User's Guide and Reference,* available on the Web at www.ibm.com/software/data/db2imstools/library.html

DB2 Connect™
- *IBM DB2 Connect Quick Beginnings for DB2 Connect Enterprise Edition*, GC09-4833
- *IBM DB2 Connect Quick Beginnings for DB2 Connect Personal Edition*, GC09-4834
- *IBM DB2 Connect User's Guide*, SC09-4835

DB2 DataPropagator™
- *DB2 Universal Database Replication Guide and Reference*, SC27-1121

DB2 Data Encryption for IMS and DB2 Databases
- *IBM Data Encryption for IMS and DB2 Databases User's Guide*, SC18-7336

DB2 Performance Expert for z/OS, Version 1

The following books are part of the DB2 Performance Expert library. Some of these books include information about the following tools: IBM DB2 Performance Expert for z/OS; IBM DB2 Performance Monitor for z/OS; and DB2 Buffer Pool Analyzer for z/OS.

- *DB2 Performance Expert for z/OS Buffer Pool Analyzer User's Guide*, SC18-7972
- *DB2 Performance Expert for z/OS and Multiplatforms Installation and Customization*, SC18-7973
- *DB2 Performance Expert for z/OS Messages*, SC18-7974
- *DB2 Performance Expert for z/OS Monitoring Performance from ISPF*, SC18-7975

- *DB2 Performance Expert for z/OS and Multiplatforms Monitoring Performance from the Workstation*, SC18-7976
- *DB2 Performance Expert for z/OS Program Directory*, GI10-8549
- *DB2 Performance Expert for z/OS Report Command Reference*, SC18-7977
- *DB2 Performance Expert for z/OS Report Reference*, SC18-7978
- *DB2 Performance Expert for z/OS Reporting User's Guide*, SC18-7979

DB2 Query Management Facility (QMF) Version 8.1
- *DB2 Query Management Facility: DB2 QMF High Performance Option User’s Guide for TSO/CICS*, SC18-7450
- *DB2 Query Management Facility: DB2 QMF Messages and Codes*, GC18-7447
- *DB2 Query Management Facility: DB2 QMF Reference*, SC18-7446
- *DB2 Query Management Facility: Developing DB2 QMF Applications*, SC18-7651
- *DB2 Query Management Facility: Getting Started with DB2 QMF for Windows and DB2 QMF for WebSphere*, SC18-7449
- *DB2 Query Management Facility: Installing and Managing DB2 QMF for TSO/CICS*, GC18-7444
- *DB2 Query Management Facility: Installing and Managing DB2 QMF for Windows and DB2 QMF for WebSphere*, GC18-7448
- *DB2 Query Management Facility: Introducing DB2 QMF*, GC18-7443
- *DB2 Query Management Facility: Using DB2 QMF*, SC18-7445
- *DB2 Query Management Facility: DB2 QMF Visionary Developer's Guide*, SC18-9093
- *DB2 Query Management Facility: DB2 QMF Visionary Getting Started Guide*, GC18-9092

DB2 Redbooks™

For access to all IBM Redbooks about DB2, see the IBM Redbooks Web page at www.ibm.com/redbooks

DB2 Server for VSE & VM
- *DB2 Server for VM: DBS Utility*, SC09-2983

DB2 Universal Database Cross-Platform information
- *IBM DB2 Universal Database SQL Reference for Cross-Platform Development*, available at www.ibm.com/software/data/developer/cpsqlref/

DB2 Universal Database for iSeries

The following books are available at www.ibm.com/iseries/infocenter
- *DB2 Universal Database for iSeries Performance and Query Optimization*
- *DB2 Universal Database for iSeries Database Programming*
- *DB2 Universal Database for iSeries SQL Programming Concepts*
- *DB2 Universal Database for iSeries SQL Programming with Host Languages*
- *DB2 Universal Database for iSeries SQL Reference*
- *DB2 Universal Database for iSeries Distributed Data Management*
- *DB2 Universal Database for iSeries Distributed Database Programming*

DB2 Universal Database for Linux, UNIX, and Windows:
- *DB2 Universal Database Administration Guide: Planning*, SC09-4822
- *DB2 Universal Database Administration Guide: Implementation*, SC09-4820
- *DB2 Universal Database Administration Guide: Performance*, SC09-4821
- *DB2 Universal Database Administrative API Reference*, SC09-4824
- *DB2 Universal Database Application Development Guide: Building and Running Applications*, SC09-4825
- *DB2 Universal Database Call Level Interface Guide and Reference, Volumes 1 and 2*, SC09-4849 and SC09-4850
- *DB2 Universal Database Command Reference*, SC09-4828
- *DB2 Universal Database SQL Reference Volume 1*, SC09-4844
- *DB2 Universal Database SQL Reference Volume 2*, SC09-4845

Device Support Facilities
- *Device Support Facilities User's Guide and Reference*, GC35-0033

DFSMS

These books provide information about a variety of components of DFSMS, including z/OS DFSMS, z/OS DFSMSdfp™, z/OS DFSMSdss, z/OS DFSMShsm, and z/OS DFP.
- *z/OS DFSMS Access Method Services for Catalogs*, SC26-7394
- *z/OS DFSMSdss Storage Administration Guide*, SC35-0423

- *z/OS DFSMSdss Storage Administration Reference*, SC35-0424
- *z/OS DFSMShsm Managing Your Own Data*, SC35-0420
- *z/OS DFSMSdfp: Using DFSMSdfp in the z/OS Environment*, SC26-7473
- *z/OS DFSMSdfp Diagnosis Reference*, GY27-7618
- *z/OS DFSMS: Implementing System-Managed Storage*, SC27-7407
- *z/OS DFSMS: Macro Instructions for Data Sets*, SC26-7408
- *z/OS DFSMS: Managing Catalogs*, SC26-7409
- *z/OS DFSMS: Program Management*, SA22-7643
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
- *z/OS DFSMSdfp Storage Administration Reference*, SC26-7402
- *z/OS DFSMS: Using Data Sets*, SC26-7410
- *DFSMS/MVS: Using Advanced Services* , SC26-7400
- *DFSMS/MVS: Utilities*, SC26-7414

DFSORT™
- *DFSORT Application Programming: Guide*, SC33-4035
- *DFSORT Installation and Customization*, SC33-4034

Distributed Relational Database Architecture
- *Open Group Technical Standard*; the Open Group presently makes the following DRDA books available through its Web site at www.opengroup.org
  – *Open Group Technical Standard, DRDA Version 3 Vol. 1: Distributed Relational Database Architecture*
  – *Open Group Technical Standard, DRDA Version 3 Vol. 2: Formatted Data Object Content Architecture*
  – *Open Group Technical Standard, DRDA Version 3 Vol. 3: Distributed Data Management Architecture*

Domain Name System
- *DNS and BIND, Third Edition, Paul Albitz and Cricket Liu, O'Reilly*, ISBN 0-59600-158-4

Education
- Information about IBM educational offerings is available on the Web at www.ibm.com/software/info/education/
- A collection of glossaries of IBM terms is available on the IBM Terminology Web site at www.ibm.com/ibm/terminology/index.html

eServer zSeries®
- *IBM eServer zSeries Processor Resource/System Manager Planning Guide*, SB10-7033

Fortran: VS Fortran
- *VS Fortran Version 2: Language and Library Reference*, SC26-4221
- *VS Fortran Version 2: Programming Guide for CMS and MVS*, SC26-4222

High Level Assembler
- *High Level Assembler for MVS and VM and VSE Language Reference*, SC26-4940
- *High Level Assembler for MVS and VM and VSE Programmer's Guide*, SC26-4941

ICSF
- *z/OS ICSF Overview*, SA22-7519
- *Integrated Cryptographic Service Facility Administrator's Guide*, SA22-7521

IMS Version 8

IMS product information is available on the IMS Library Web page, which you can find at www.ibm.com/ims
- *IMS Administration Guide: System*, SC27-1284
- *IMS Administration Guide: Transaction Manager*, SC27-1285
- *IMS Application Programming: Database Manager*, SC27-1286
- *IMS Application Programming: Design Guide*, SC27-1287
- *IMS Application Programming: Transaction Manager*, SC27-1289
- *IMS Command Reference*, SC27-1291
- *IMS Customization Guide*, SC27-1294
- *IMS Install Volume 1: Installation Verification*, GC27-1297
- *IMS Install Volume 2: System Definition and Tailoring*, GC27-1298
- *IMS Messages and Codes Volumes 1 and 2*, GC27-1301 and GC27-1302
- *IMS Utilities Reference: System*, SC27-1309

General information about IMS Batch Terminal Simulator for z/OS is available on the Web at www.ibm.com/software/data/db2imstools/library.html

IMS DataPropagator
- *IMS DataPropagator for z/OS Administrator's Guide for Log*, SC27-1216
- *IMS DataPropagator: An Introduction*, GC27-1211

- *IMS DataPropagator for z/OS Reference*, SC27-1210

ISPF

- *z/OS ISPF Dialog Developer's Guide*, SC23-4821
- *z/OS ISPF Messages and Codes*, SC34-4815
- *z/OS ISPF Planning and Customizing*, GC34-4814
- *z/OS ISPF User's Guide Volumes 1 and 2*, SC34-4822 and SC34-4823

Java for z/OS

- *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201

Language Environment

- *Debug Tool User's Guide and Reference*, SC18-7171
- *Debug Tool for z/OS and OS/390 Reference and Messages*, SC18-7172
- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562

MQSeries®

- *MQSeries Application Messaging Interface*, SC34-5604
- *MQSeries for OS/390 Concepts and Planning Guide*, GC34-5650
- *MQSeries for OS/390 System Setup Guide*, SC34-5651

National Language Support

- *National Language Design Guide Volume 1*, SE09-8001
- *IBM National Language Support Reference Manual Volume 2*, SE09-8002

NetView®

- *Tivoli NetView for z/OS Installation: Getting Started*, SC31-8872
- *Tivoli NetView for z/OS User's Guide*, GC31-8849

Microsoft ODBC

Information about Microsoft ODBC is available at http://msdn.microsoft.com/library/

Parallel Sysplex Library

- *System/390 9672 Parallel Transaction Server, 9672 Parallel Enterprise Server, 9674 Coupling Facility System Overview For R1/R2/R3 Based Models*, SB10-7033
- *z/OS Parallel Sysplex Application Migration*, SA22-7662
- *z/OS Parallel Sysplex Overview: An Introduction to Data Sharing and Parallelism*, SA22-7661
- *z/OS Parallel Sysplex Test Report*, SA22-7663

The *Parallel Sysplex Configuration Assistant* is available at www.ibm.com/s390/pso/psotool

PL/I: Enterprise PL/I for z/OS and OS/390

- *IBM Enterprise PL/I for z/OS and OS/390 Language Reference*, SC27-1460
- *IBM Enterprise PL/I for z/OS and OS/390 Programming Guide*, SC27-1457

PL/I: OS PL/I

- *OS PL/I Programming Guide*, SC26-4307

SMP/E

- *SMP/E for z/OS and OS/390 Reference*, SA22-7772
- *SMP/E for z/OS and OS/390 User's Guide*, SA22-7773

Storage Management

- *z/OS DFSMS: Implementing System-Managed Storage*, SC26-7407
- *MVS/ESA Storage Management Library: Managing Data*, SC26-7397
- *MVS/ESA Storage Management Library: Managing Storage Groups*, SC35-0421
- *MVS Storage Management Library: Storage Management Subsystem Migration Planning Guide*, GC26-7398

System Network Architecture (SNA)

- *SNA Formats*, GA27-3136
- *SNA LU 6.2 Peer Protocols Reference*, SC31-6808
- *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, GC30-3084
- *SNA/Management Services Alert Implementation Guide*, GC31-6809

TCP/IP

- *IBM TCP/IP for MVS: Customization & Administration Guide*, SC31-7134
- *IBM TCP/IP for MVS: Diagnosis Guide*, LY43-0105
- *IBM TCP/IP for MVS: Messages and Codes*, SC31-7132

- *IBM TCP/IP for MVS: Planning and Migration Guide*, SC31-7189

TotalStorage® Enterprise Storage Server
- *RAMAC Virtual Array: Implementing Peer-to-Peer Remote Copy*, SG24-5680
- *Enterprise Storage Server Introduction and Planning*, GC26-7444
- *IBM RAMAC Virtual Array*, SG24-6424

Unicode
- *z/OS Support for Unicode: Using Conversion Services*, SA22-7649

Information about Unicode, the Unicode consortium, the Unicode standard, and standards conformance requirements is available at www.unicode.org

VTAM
- *Planning for NetView, NCP, and VTAM*, SC31-8063
- *VTAM for MVS/ESA Diagnosis*, LY43-0078
- *VTAM for MVS/ESA Messages and Codes*, GC31-8369
- *VTAM for MVS/ESA Network Implementation Guide*, SC31-8370
- *VTAM for MVS/ESA Operation*, SC31-8372
- *VTAM for MVS/ESA Programming*, SC31-8373
- *VTAM for MVS/ESA Programming for LU 6.2*, SC31-8374
- *VTAM for MVS/ESA Resource Definition Reference*, SC31-8377

WebSphere® family
- *WebSphere MQ Integrator Broker: Administration Guide*, SC34-6171
- *WebSphere MQ Integrator Broker for z/OS: Customization and Administration Guide*, SC34-6175
- *WebSphere MQ Integrator Broker: Introduction and Planning*, GC34-5599
- *WebSphere MQ Integrator Broker: Using the Control Center*, SC34-6168

z/Architecture™
- *z/Architecture Principles of Operation*, SA22-7832

z/OS
- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS Communications Server: IP Configuration Guide*, SC31-8875
- *z/OS DCE Administration Guide*, SC24-5904
- *z/OS DCE Introduction*, GC24-5911
- *z/OS DCE Messages and Codes*, SC24-5912
- *z/OS Information Roadmap*, SA22-7500
- *z/OS Introduction and Release Guide*, GA22-7502
- *z/OS JES2 Initialization and Tuning Guide*, SA22-7532
- *z/OS JES3 Initialization and Tuning Guide*, SA22-7549
- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562
- *z/OS Managed System Infrastructure for Setup User's Guide*, SC33-7985
- *z/OS MVS Diagnosis: Procedures*, GA22-7587
- *z/OS MVS Diagnosis: Reference*, GA22-7588
- *z/OS MVS Diagnosis: Tools and Service Aids*, GA22-7589
- *z/OS MVS Initialization and Tuning Guide*, SA22-7591
- *z/OS MVS Initialization and Tuning Reference*, SA22-7592
- *z/OS MVS Installation Exits*, SA22-7593
- *z/OS MVS JCL Reference*, SA22-7597
- *z/OS MVS JCL User's Guide*, SA22-7598
- *z/OS MVS Planning: Global Resource Serialization*, SA22-7600
- *z/OS MVS Planning: Operations*, SA22-7601
- *z/OS MVS Planning: Workload Management*, SA22-7602
- *z/OS MVS Programming: Assembler Services Guide*, SA22-7605
- *z/OS MVS Programming: Assembler Services Reference, Volumes 1 and 2*, SA22-7606 and SA22-7607
- *z/OS MVS Programming: Authorized Assembler Services Guide*, SA22-7608
- *z/OS MVS Programming: Authorized Assembler Services Reference Volumes 1-4*, SA22-7609, SA22-7610, SA22-7611, and SA22-7612
- *z/OS MVS Programming: Callable Services for High-Level Languages*, SA22-7613
- *z/OS MVS Programming: Extended Addressability Guide*, SA22-7614
- *z/OS MVS Programming: Sysplex Services Guide*, SA22-7617
- *z/OS MVS Programming: Sysplex Services Reference*, SA22-7618

- *z/OS MVS Programming: Workload Management Services*, SA22-7619
- *z/OS MVS Recovery and Reconfiguration Guide*, SA22-7623
- *z/OS MVS Routing and Descriptor Codes*, SA22-7624
- *z/OS MVS Setting Up a Sysplex*, SA22-7625
- *z/OS MVS System Codes* SA22-7626
- *z/OS MVS System Commands*, SA22-7627
- *z/OS MVS System Messages Volumes 1-10*, SA22-7631, SA22-7632, SA22-7633, SA22-7634, SA22-7635, SA22-7636, SA22-7637, SA22-7638, SA22-7639, and SA22-7640
- *z/OS MVS Using the Subsystem Interface*, SA22-7642
- *z/OS Planning for Multilevel Security*, SA22-7509
- *z/OS RMF User's Guide*, SC33-7990
- *z/OS Security Server Network Authentication Server Administration*, SC24-5926
- *z/OS Security Server RACF Auditor's Guide*, SA22-7684
- *z/OS Security Server RACF Command Language Reference*, SA22-7687
- *z/OS Security Server RACF Macros and Interfaces*, SA22-7682
- *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683
- *z/OS Security Server RACF System Programmer's Guide*, SA22-7681
- *z/OS Security Server RACROUTE Macro Reference*, SA22-7692
- *z/OS Support for Unicode: Using Conversion Services*, SA22-7649
- *z/OS TSO/E CLISTs*, SA22-7781
- *z/OS TSO/E Command Reference*, SA22-7782
- *z/OS TSO/E Customization*, SA22-7783
- *z/OS TSO/E Messages*, SA22-7786
- *z/OS TSO/E Programming Guide*, SA22-7788
- *z/OS TSO/E Programming Services*, SA22-7789
- *z/OS TSO/E REXX Reference*, SA22-7790
- *z/OS TSO/E User's Guide*, SA22-7794
- *z/OS UNIX System Services Command Reference*, SA22-7802
- *z/OS UNIX System Services Messages and Codes*, SA22-7807
- *z/OS UNIX System Services Planning*, GA22-7800
- *z/OS UNIX System Services Programming: Assembler Callable Services Reference*, SA22-7803
- *z/OS UNIX System Services User's Guide*, SA22-7801

z/OS mSys for Setup

- *z/OS Managed System Infrastructure for Setup DB2 Customization Center User's Guide*, available in softcopy format at www.ibm.com/db2/zos/v8books.html
- *z/OS Managed System Infrastructure for Setup User's Guide*, SC33-7985

Index

Readers' Comments — We'd Like to Hear from You

DB2 Universal Database for z/OS
ODBC Guide and Reference
Version 8

Publication No. SC18-7423-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC18-7423-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines
Corporation
H150/090
555 Bailey Avenue
San Jose, CA 95141-9989
U. S. A.

Fold and Tape **Please do not staple** Fold and Tape

SC18-7423-00

Cut or Fold
Along Line

Program Number: 5625-DB2

Printed in USA

SC18-7423-00

Spine information:

IBM DB2 Universal Database for z/OS **Version 8** **ODBC Guide and Reference**