Db2 11 for z/OS

# *ODBC Guide and Reference*



SC19-4063-05

**Notes**

Before using this information and the product it supports, be sure to read the general information under "Notices" at the end of this information.

Subsequent editions of this PDF will not be delivered in IBM Publications Center. Always download the latest edition from PDF format manuals for Db2 11 for z/OS (Db2 Home).

**2020-06-17 edition**

This edition applies to Db2® 11 for z/OS® (product number 5615-DB2), Db2 11 for z/OS Value Unit Edition (product number 5697-P43), 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.

**© Copyright International Business Machines Corporation 1997, 2020.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents


**About this information..... ix**
Who should read this information.....ix
Db2 Utilities Suite for z/OS.....ix
Terminology and citations.....x
Accessibility features for Db2 11 for z/OS.....x
How to send your comments..... xi
How to read syntax diagrams.....xi

**Chapter 1. Introduction to Db2 ODBC..... 1**
Db2 ODBC background information..... 1
Differences between Db2 ODBC and ODBC 3.0..... 2
Db2 ODBC support for ODBC features.....2
Differences between Db2 ODBC and embedded SQL..... 4
Advantages of using Db2 ODBC.....7
Considerations for choosing between SQL and Db2 ODBC..... 7

**Chapter 2. Conceptual view of a Db2 ODBC application.....9**
Initialization and termination of an ODBC program.....10
Handles.....10
ODBC connection model..... 11
How to specify the connection type..... 12
How to connect to one or more data sources..... 13
Transaction processing in Db2 ODBC..... 14
Statement handle allocation.....15
Preparation and execution of SQL statements.....16
How an ODBC program processes results.....18
Commit and rollback in Db2 ODBC..... 21
Function for freeing statement handles..... 22
Diagnostics.....23
Function return codes..... 23
SQLSTATEs for ODBC error reporting..... 24
SQLCA retrieval in an ODBC application..... 25
Data types and data conversion..... 25
C and SQL data types..... 25
C data types that do not map to SQL data types..... 30
Data conversion.....31
Characteristics of string arguments..... 34
Length of string arguments..... 34
Nul-termination of strings.....34
String truncation.....35
Interpretation of strings.....35
Functions for querying environment and data source information.....35

**Chapter 3. Configuring Db2 ODBC and running sample applications..... 37**
Running the SMP/E jobs for Db2 ODBC installation.....37
The Db2 ODBC run time environment.....37
Connectivity requirements.....38
Extra performance linkage.....39
64-bit ODBC driver..... 39
Db2 ODBC run time environment setup.....39

Binding DBRMs to create packages....40
Binding the application plan....42
Setting up Db2 ODBC for the z/OS UNIX environment....43
Overview of preparing and executing a Db2 ODBC application....44
Db2 ODBC application requirements....45
Preparing and executing an ODBC application....46
Compiling an ODBC application....47
Prelinking and link-editing an ODBC application....50
Executing an ODBC application....53
How to define a subsystem to Db2 ODBC....54
Db2 ODBC initialization file....54
How to use the initialization file....54
Db2 ODBC initialization keywords....56
Database metadata stored procedures....75
Migrating to the Db2 11 ODBC driver....75
Migrating an ODBC 31-bit application to a 64-bit application....76
Example 64-bit ODBC application....76

**Chapter 4. ODBC functions....79**
Status of support for ODBC functions....80
SQLAllocConnect() - Allocate a connection handle....86
SQLAllocEnv() - Allocate an environment handle....87
SQLAllocHandle() - Allocate a handle....87
SQLAllocStmt() - Allocate a statement handle....92
SQLBindCol() - Bind a column to an application variable....92
SQLBindFileToCol() - Associate a column with a file reference....99
SQLBindFileToParam() - Bind a parameter marker to a file reference....103
SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator....106
SQLBulkOperations() - Add, update, delete or fetch a set of rows....121
SQLCancel() - Cancel statement....125
SQLCloseCursor() - Close a cursor and discard pending results....127
SQLColAttribute() - Get column attributes....129
SQLColAttributes() - Get column attributes....137
SQLColumnPrivileges() - Get column privileges....138
SQLColumns() - Get column information....142
SQLConnect() - Connect to a data source....149
SQLDataSources() - Get a list of data sources....154
SQLDescribeCol() - Describe column attributes....157
SQLDescribeParam() - Describe parameter marker....162
SQLDisconnect() - Disconnect from a data source....165
SQLDriverConnect() - Use a connection string to connect to a data source....167
SQLEndTran() - End transaction of a connection....172
SQLError() - Retrieve error information....175
SQLExecDirect() - Execute a statement directly....177
SQLExecute() - Execute a statement....183
SQLExtendedFetch() - Fetch an array of rows....186
SQLFetch() - Fetch the next row....192
SQLFetchScroll() - Fetch the next row....198
SQLForeignKeys() - Get a list of foreign key columns....206
SQLFreeConnect() - Free a connection handle....213
SQLFreeEnv() - Free an environment handle....214
SQLFreeHandle() - Free a handle....215
SQLFreeStmt() - Free (or reset) a statement handle....217
SQLGetConnectAttr() - Get current attribute setting....220
SQLGetConnectOption() - Return current setting of a connect option....223
SQLGetCursorName() - Get cursor name....223
SQLGetData() - Get data from a column....228

SQLGetDiagRec() - Get multiple field settings of diagnostic record ... 239
SQLGetEnvAttr() - Return current setting of an environment attribute ... 242
SQLGetFunctions() - Get functions ... 244
SQLGetInfo() - Get general information ... 249
SQLGetLength() - Retrieve length of a string value ... 276
SQLGetPosition() - Find the starting position of a string ... 278
SQLGetSQLCA() - Get SQLCA data structure ... 283
SQLGetStmtAttr() - Get current setting of a statement attribute ... 287
SQLGetStmtOption() - Return current setting of a statement option ... 291
SQLGetSubString() - Retrieve a portion of a string value ... 291
SQLGetTypeInfo() - Get data type information ... 295
SQLMoreResults() - Check for more result sets ... 303
SQLNativeSql() - Get native SQL text ... 306
SQLNumParams() - Get number of parameters in an SQL statement ... 309
SQLNumResultCols() - Get number of result columns ... 310
SQLParamData() - Get next parameter for which a data value is needed ... 313
SQLParamOptions() - Specify an input array for a parameter ... 315
SQLPrepare() - Prepare a statement ... 317
SQLPrimaryKeys() - Get primary key columns of a table ... 323
SQLProcedureColumns() - Get procedure input/output parameter information ... 327
SQLProcedures() - Get a list of procedure names ... 336
SQLPutData() - Pass a data value for a parameter ... 340
SQLRowCount() - Get row count ... 343
SQLSetColAttributes() - Set column attributes ... 346
SQLSetConnectAttr() - Set connection attributes ... 350
SQLSetConnection() - Set connection handle ... 362
SQLSetConnectOption() - Set connection option ... 363
SQLSetCursorName() - Set cursor name ... 364
SQLSetEnvAttr() - Set environment attributes ... 367
SQLSetParam() - Bind a parameter marker to a buffer ... 371
SQLSetPos - Set the cursor position in a rowset ... 372
SQLSetStmtAttr() - Set statement attributes ... 378
SQLSetStmtOption() - Set statement attribute ... 389
SQLSpecialColumns() - Get special (row identifier) columns ... 390
SQLStatistics() - Get index and statistics information for a base table ... 396
SQLTablePrivileges() - Get table privileges ... 401
SQLTables() - Get table information ... 405
SQLTransact() - Transaction management ... 410

**Chapter 5. Advanced features ... 411**
Functions for setting and retrieving environment, connection, and statement attributes ... 411
Functions for setting and retrieving environment attributes ... 413
Functions for setting and retrieving connection attributes ... 413
Functions for setting and retrieving statement attributes ... 413
ODBC and distributed units of work ... 414
Functions for establishing a distributed unit-of-work connection ... 414
Coordinated connections in a Db2 ODBC application ... 416
Global transactions in ODBC programs ... 418
Use of ODBC for querying the Db2 catalog ... 420
Catalog query functions ... 420
The Db2 ODBC shadow catalog ... 423
Using arrays to pass parameter values ... 424
Retrieval of a result set into an array ... 428
Column-wise binding for array data ... 429
Row-wise binding for array data ... 430
The ODBC row status array ... 432
Column-wise and row-wise binding example ... 433

ODBC limited block fetch ... 434
Scrollable cursors in Db2 ODBC ... 435
Scrollable cursor characteristics in Db2 ODBC ... 435
Relative and absolute scrolling in Db2 ODBC applications ... 436
Steps for retrieving data with scrollable cursors in a Db2 ODBC application ... 438
ODBC scrollable cursor example ... 440
Performing bulk inserts with SQLBulkOperations() ... 447
Updates to Db2 tables with SQLSetPos() ... 448
Updating rows in a rowset with SQLSetPos() ... 448
Deleting rows in a rowset with SQLSetPos() ... 450
Input and retrieval of long data in pieces ... 451
Providing long data for bulk inserts and positioned updates ... 453
Examples of using decimal floating point data in an ODBC application ... 454
Variable-length timestamps in ODBC applications ... 455
Using LOBs ... 458
Using LOB locators ... 458
LOB and LOB locator example ... 460
LOB file reference variables in ODBC applications ... 460
XML data in ODBC applications ... 461
XML column updates in ODBC applications ... 461
XML data retrieval in ODBC applications ... 463
Distinct types in Db2 ODBC applications ... 465
Stored procedures for ODBC applications ... 466
Advantages of using stored procedures ... 467
Stored procedure calls in a Db2 ODBC application ... 467
Rules for a Db2 ODBC stored procedure ... 468
Result sets from stored procedures in ODBC applications ... 468
Multithreaded and multiple-context applications in Db2 ODBC ... 471
Db2 ODBC support for multiple Language Environment threads ... 471
When to use multiple Language Environment threads ... 473
Db2 ODBC support of multiple contexts ... 473
External contexts ... 477
Application deadlocks ... 478
Application encoding schemes and Db2 ODBC ... 479
Types of encoding schemes ... 479
Application programming guidelines for handling different encoding schemes ... 479
Suffix-W API function syntax ... 481
Examples of handling the application encoding scheme ... 485
Embedded SQL and Db2 ODBC in the same program ... 493
Vendor escape clauses ... 495
Function for determining ODBC vendor escape clause support ... 495
Escape clause syntax ... 495
ODBC-defined SQL extensions ... 496
Extended indicators in ODBC applications ... 500
ODBC programming hints and tips ... 500
Guidelines for avoiding common problems ... 500
Techniques for improving application performance ... 501
Techniques for reducing network flow ... 504
Techniques for maximizing application portability ... 505

**Chapter 6. Problem diagnosis ... 507**
ODBC trace types ... 507
Application trace ... 507
ODBC diagnostic trace ... 510
Stored procedure trace ... 516
Abnormal termination ... 519
Internal error code ... 520

**Chapter 7. Db2 ODBC reference information..... 521**
Db2 ODBC and ODBC differences..... 521
Db2 ODBC and ODBC drivers..... 521
ODBC APIs and data types..... 522
Isolation levels..... 524
Extended scalar functions..... 524
Errors returned by extended scalar functions..... 525
String functions..... 525
Date and time functions..... 526
System functions..... 526
SQLSTATE cross reference..... 527
Data conversion between the application and the database server..... 542
SQL data type attributes..... 542
SQL to C data conversion..... 546
C to SQL data conversion..... 556
Deprecated ODBC functions..... 562
Deprecated ODBC functions and their replacements..... 562
Changes to `SQLGetInfo()` InfoType argument values..... 563
Changes to `SQLSetConnectAttr()` attributes..... 563
Changes to `SQLSetEnvAttr()` attributes..... 564
Changes to `SQLSetStmtAttr()` attributes..... 564
ODBC 3.0 driver behavior..... 565
SQLSTATE mappings..... 565
Changes to datetime data types..... 567
Example Db2 ODBC code..... 568
DSN8O3VP sample application..... 568
Client application calling a Db2 ODBC stored procedure..... 572

**Information resources for Db2 11 for z/OS and related products..... 587**

**Notices..... 589**
Programming interface information..... 590
Trademarks..... 590
Terms and conditions for product documentation..... 590
Privacy policy considerations..... 591

**Glossary..... 593**

**Index..... 595**

# About this information

This information provides the information necessary to write applications that use Db2 ODBC to access IBM® Db2 servers, or to access any database system that supports DRDA level 1 or DRDA level 2 protocols. This information should also be used as a supplement for writing portable ODBC applications that can be executed in a native environment using Db2 ODBC.

Throughout this information, "Db2" means "Db2 11 for z/OS". References to other Db2 products use complete names or specific abbreviations.

**Important:** To find the most up to date content for Db2 11 for z/OS, always use IBM Knowledge Center or download the latest PDF file from PDF format manuals for Db2 11 for z/OS (Db2 Home).

This information assumes that Db2 11 is running in new-function mode, and that your application is running with the application compatibility value of 'V11R1'.

**Availability of new function in Db2 11**

The behavior of data definition statements such as CREATE, ALTER, and DROP, which embed data manipulation SQL statements that contain new capabilities, depends on the application compatibility value that is in effect for the application. An application compatibility value of 'V11R1' must be in effect for applications to use new capability in embedded statements such as SELECT, INSERT, UPDATE, DELETE, MERGE, CALL, and SET *assignment-statement*. Otherwise, an application compatibility value of 'V10R1' can be used for data definition statements.

Generally, new SQL capabilities, including changes to existing language elements, functions, data manipulation statements, and limits, are available only in new-function mode with applications set to an application compatibility value of 'V11R1'.

Optimization and virtual storage enhancements are available in conversion mode unless stated otherwise.

SQL statements can continue to run with the same expected behavior as in DB2® 10 new-function mode with an application compatibility value of 'V10R1'.

## Who should read this information

This information is for the following users:

- Db2 for z/OS 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.

## Db2 Utilities Suite for z/OS

**Important:** In Db2 11, the Db2 Utilities Suite for z/OS 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.

Db2 11 utilities can use the DFSORT program regardless of whether you purchased a license for DFSORT on your system. For more information, see the following informational APARs:

- II14047
- II14213
- II13495

Db2 utilities can use IBM Db2 Sort for z/OS (5655-W42) as an alternative to DFSORT for utility SORT and MERGE functions. Use of Db2 Sort for z/OS requires the purchase of a Db2 Sort for z/OS license. For more information about Db2 Sort for z/OS, see Db2 Sort for z/OS.

© Copyright IBM Corp. 1997, 2020

**Related concepts**

Db2 utilities packaging (Db2 Utilities)

## Terminology and citations

When referring to a Db2 product other than Db2 for z/OS, this information uses the product's full name to avoid ambiguity.

**About the Db2 brand change:** IBM has re-branded DB2 to Db2, and Db2 for z/OS is now the name of the offering previously know as "DB2 for z/OS". During and after implementation of this change, you might sometimes still see references to the original names, such as "DB2 for z/OS" and "DB2", in different IBM web pages and documents. However, assume that these refer to the same "Db2 for z/OS" product when the PID, Entitlement Entity, version, modification, and release information match. For more information, see Revised naming for IBM Db2 family products on IBM z/OS platform.

The following terms are used as indicated:

**Db2**
Represents either the Db2 licensed program or a particular Db2 subsystem.

**Tivoli® OMEGAMON® XE for Db2 Performance Expert on z/OS**
Refers to any of the following products:

- IBM Tivoli OMEGAMON XE for Db2 Performance Expert on z/OS
- IBM Db2 Performance Monitor on z/OS
- IBM Db2 Performance Expert for Multiplatforms and Workgroups
- IBM Db2 Buffer Pool Analyzer for z/OS

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

**CICS®**
Represents CICS Transaction Server for z/OS.

**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 features for Db2 11 for z/OS

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

### Accessibility features

The following list includes the major accessibility features in z/OS products, including Db2 11 for z/OS. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers and screen magnifiers.
- Customization of display attributes such as color, contrast, and font size

**Tip:** The IBM Knowledge Center (which includes information for Db2 for z/OS) and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

### Keyboard navigation

For information about navigating the Db2 for z/OS ISPF panels using TSO/E or ISPF, refer to the *z/OS TSO/E Primer*, the *z/OS TSO/E User's Guide*, and the *z/OS ISPF User's Guide*. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

### Related accessibility information

### IBM and accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

## 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 for z/OS documentation.

Send your comments by email to db2zinfo@us.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 or a help topic title).

## How to read syntax diagrams

Certain conventions apply to the syntax diagrams that are used in IBM documentation.

Apply the following rules when reading the syntax diagrams that are used in Db2 for z/OS documentation:

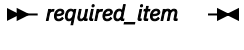
- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►—— symbol indicates the beginning of a statement.

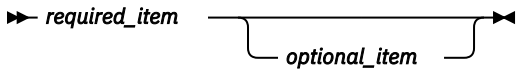
  The ——► symbol indicates that the statement syntax is continued on the next line.

  The ►—— symbol indicates that a statement is continued from the previous line.

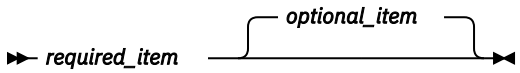
  The ——►◄ symbol indicates the end of a statement.
- Required items appear on the horizontal line (the main path).

- Optional items appear below the main path.

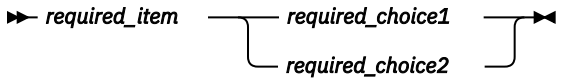
  
  

  If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

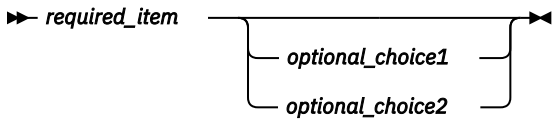
  
  

- If you can choose from two or more items, they appear vertically, in a stack.

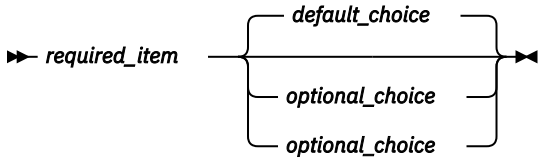
  If you *must* choose one of the items, one item of the stack appears on the main path.

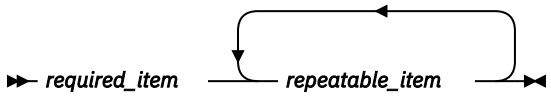
  If choosing one of the items is optional, the entire stack appears below the main path.



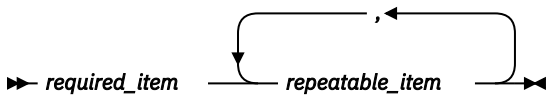

If one of the items is the default, it appears above the main path and the remaining choices are shown below.




- An arrow returning to the left, above the main line, indicates an item that can be repeated.

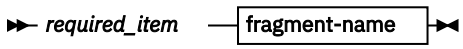



If the repeat arrow contains a comma, you must separate repeated items with a comma.

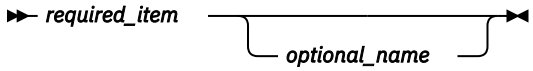



A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Sometimes a diagram must be split into fragments. The syntax fragment is shown separately from the main syntax diagram, but the contents of the fragment should be read as if they are on the main path of the diagram.




**fragment-name**




- With the exception of XPath keywords, keywords appear in uppercase (for example, FROM). Keywords must be spelled exactly as shown. XPath keywords are defined as lowercase names, and must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

**Related concepts**

Commands in Db2 (Db2 Commands)
Db2 online utilities (Db2 Utilities)
Db2 stand-alone utilities (Db2 Utilities)

# Chapter 1. Introduction to Db2 ODBC

Db2 ODBC offers advantages over SQL and provides helpful extensions for application programming.

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.

**Related concepts**

Advantages of using Db2 ODBC
Db2 ODBC provides a number of key features that offer advantages in contrast to embedded SQL.

Db2 ODBC background information
The Db2 ODBC interface allows you to create portable applications. The interface also allows you to load drivers dynamically at run time.

Differences between Db2 ODBC and embedded SQL
Even though key differences exist between Db2 ODBC and embedded SQL, Db2 ODBC can execute any SQL statements that can be prepared dynamically in embedded SQL.

Considerations for choosing between SQL and Db2 ODBC
Before you determine which interface to use, consider key factors like performance, encapsulation, and security.

## Db2 ODBC background information

The Db2 ODBC interface allows you to create portable applications. The interface also allows you to load drivers dynamically at run time.

To understand Db2 ODBC or any callable SQL interface, you should 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 database management system's library. The driver manager mediates the application's function calls at run time and ensures they are directed to the appropriate ODBC driver.

The ODBC driver manager only knows about the ODBC-specific functions, that is, those functions supported by the database management system for which no API is specified. Therefore, functions that are specific to one database management system cannot be directly accessed in an ODBC environment. However, dynamic SQL statements that are specific to a database management system are indirectly supported using a mechanism called the vendor escape clause.

© Copyright IBM Corp. 1997, 2020

ODBC is not limited to Microsoft operating systems. Other implementations are available, such as Db2 ODBC, or are emerging on various platforms.

**Related concepts**

Vendor escape clauses
Vendor escape clauses increase the portability of your application if your application accesses multiple data sources from different vendors. However, if your application accesses only Db2 data sources, you have no reason to use vendor escape clauses.

## Differences between Db2 ODBC and ODBC 3.0

Several differences exist between the drivers and runtime environment of Db2 ODBC and ODBC 3.0.

Db2 ODBC is derived from the ISO Call Level Interface Draft International Standard (ISO CLI DIS) and ODBC 3.0.

If you port existing ODBC applications to Db2 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 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.

The Db2 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 for Linux, UNIX, and Windows is referred to as the "Call Level Interface."

The use of Db2 ODBC in this publication refers to Db2 for z/OS support of Db2 ODBC unless otherwise noted.

**Related concepts**

Db2 ODBC and ODBC drivers
Differences exist between the Db2 ODBC and ODBC drivers. Generally, Db2 ODBC supports a subset of the functions that the ODBC driver provides.

The Db2 ODBC run time environment
Db2 ODBC support is implemented as an IBM C/C++ Dynamic Load Library (DLL). All API calls are routed through the single ODBC driver that is loaded at run time into the application address space.

**Related information**

Microsoft open database connectivity (ODBC)

## Db2 ODBC support for ODBC features

Db2 ODBC supports ODBC 3.0 features with certain exceptions.

Db2 ODBC support should be viewed as consisting of most of ODBC 3.0 along with 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 for z/OS ODBC environment.
- ODBC level 1 conformance with the following exceptions:
  - Asynchronous execution of ODBC functions for individual connections is not supported.

  - 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()`.
  - 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

.

The following Db2 features are available to both ODBC and Db2 ODBC applications:

- The double-byte (graphic) data types
- Stored procedures
- Distributed unit of work (DUW) as defined by DRDA, two-phase commit
- Distinct types
- User-defined functions
- Unicode and ASCII support

Db2 ODBC contains extensions to access Db2 features that can not be accessed by ODBC applications:

- SQLCA access for detailed Db2 specific diagnostic information
- Control over nul-termination of output strings
- Support for large objects (LOBs) and LOB locators

**Related concepts**

ODBC and distributed units of work
You can write Db2 ODBC applications to use distributed units of work.

Stored procedures for ODBC applications
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.

Length of string arguments
String arguments for both output and input have associated length arguments. You should always use a valid output length argument.

Using LOBs
The term large object (LOB) refers to any type of large object. Db2 supports three LOB data types: binary large object (BLOB), character large object (CLOB), and double-byte character large object (DBCLOB).

Application encoding schemes and Db2 ODBC
Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The Db2 ODBC driver supports input and output character string arguments to ODBC APIs and input and output host variable data in each of these encoding schemes.

Distinct types in Db2 ODBC applications
You can define your own SQL data type, which is 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*.

**Related reference**

SQLGetSQLCA() - Get SQLCA data structure

`SQLGetSQLCA()` returns the SQLCA (SQL communication area) that is associated with preparing and executing an SQL statement, fetching data, or closing a cursor. The SQLCA can return supplemental information about the data that is obtained by `SQLGetDiagRec()`.

Status of support for ODBC functions
Each function has its own ODBC 3.0 conformance level, and Db2 ODBC support level, and certain functions are deprecated.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

Db2 ODBC and ODBC differences
Db2 ODBC and standard ODBC differ in several ways for their drivers, data types, and isolation levels.

## Differences between Db2 ODBC and embedded SQL

Even though key differences exist between Db2 ODBC and embedded SQL, Db2 ODBC can execute any SQL statements that can be prepared dynamically in 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).

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 on page 5 lists each Db2 for z/OS SQL statement and indicates whether you can execute that statement with Db2 ODBC.

Each database management system might have additional statements that can be dynamically prepared, in which case Db2 ODBC passes them to the database management system.

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

*Table 1. ODBC support for 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] |
| 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 |

*Table 1. ODBC support for SQL statements (continued)*

| SQL statement | Dynamic[1] | Db2 ODBC[2] |
|---|---|---|
| INSERT | Yes | Yes |
| LABEL ON | Yes | Yes |
| LOCK TABLE | Yes | Yes |
| MERGE"5" on page 6 | Yes | Yes |
| OPEN | No | SQLExecute(), SQLExecDirect() |
| PREPARE | No | SQLPrepare() |
| RELEASE | No | No |
| RENAME | Yes | Yes |
| 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 |
| SET CURRENT PACKAGESET | No | No |
| SET CURRENT PATH | Yes | Yes |
| SET CURRENT SCHEMA | Yes | Yes |
| SET CURRENT SQLID | Yes | Yes |
| UPDATE | Yes | Yes |
| WHENEVER[3] | No | No |

**Note:**

1. All statements in this list can be coded as static SQL, but only those marked Yes 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().
5. The FOR *n* ROWS clause cannot be specified in a MERGE statement in a Db2 ODBC program. To specify the number of rows to be merged, use SQLSetStmtAttr() with the SQL_ATTR_PARAMSET_SIZE statement attribute.

**Related reference**

SQLSTATE cross reference

SQLSTATEs are returned by the Db2 ODBC application as a diagnostic tool for encountered errors, indicating the cause for these errors.

## Advantages of using Db2 ODBC

Db2 ODBC provides a number of key features that offer advantages in contrast to embedded SQL.

Db2 ODBC has the following features:

- 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 database management system catalog tables. The result sets that are returned are consistent across database management systems. 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.

## Considerations for choosing between SQL and Db2 ODBC

Before you determine which interface to use, consider key factors like performance, encapsulation, and security.

**Introductory concepts**

Submitting SQL statements to Db2 (Introduction to Db2 for z/OS)
Dynamic SQL applications (Introduction to Db2 for z/OS)
Use of ODBC to execute dynamic SQL (Introduction to Db2 for z/OS)

Db2 ODBC is ideally suited for query-based applications that require portability. Use the following information to help you decide which interface meets your needs.

**ODBC is a dynamic SQL interface**

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.

**ODBC applications can call a stored procedures that use static SQL**

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.

**An ODBC application can mix static and dynamic SQL:**

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.

**Related concepts**

Embedded SQL and Db2 ODBC in the same program
You can combine embedded static SQL with Db2 ODBC to write a mixed application.For 64-bit applications, you cannot use embedded static SQL statements.

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.

Differences between static and dynamic SQL (Db2 Application programming and SQL)

# Chapter 2. Conceptual view of a Db2 ODBC application

A typical Db2 ODBC application includes initialization, transaction processing, and termination tasks.

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.

Every Db2 ODBC application performs three core tasks: initialization, transaction processing, and termination. The following figure illustrates an ODBC application in terms of these tasks.

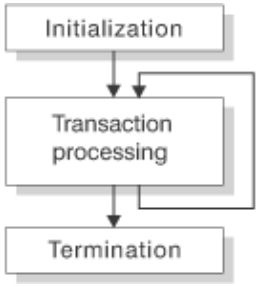



*Figure 1. Conceptual view of a Db2 ODBC application*

**Initialization**
This task allocates and initializes some resources in preparation for the transaction processing task.

**Transaction processing**
This task provides functionality to the application. It passes SQL statements to Db2 ODBC that query and modify data.

**Termination**
This task frees allocated resources. The resources generally consist of data areas identified by unique handles.

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

**Related concepts**

Functions for querying environment and data source information
Db2 ODBC provides functions that let applications retrieve information about the characteristics and capabilities of the current ODBC driver or the data source to which it is connected.

Advanced features
Db2 ODBC provides advanced features for performing setting and retrieving attributes, working with global transactions, querying the catalog, and using LOBs, XML documents, and distinct types.

Diagnostics
Diagnostics deal with warning or error conditions that are generated within an application.

Data types and data conversion
When you write a Db2 ODBC application, you must work with both SQL data types and C data types. The database server uses SQL data types, and the application uses C data types.

Initialization and termination of an ODBC program
Initialization and termination processes allocate and free resources by using handles.

Transaction processing in Db2 ODBC
Transaction processing is the second core task after initialization. During this task, SQL statements query and modify data in Db2 ODBC.

Characteristics of string arguments

© Copyright IBM Corp. 1997, 2020

String arguments in Db2 ODBC rely on conventions.

**Related information**

ODBC functions
Db2 ODBC provides various SQL-related functions with unique purposes, diagnostics, and restrictions.

# Initialization and termination of an ODBC program

Initialization and termination processes allocate and free resources by using handles.

The following figure shows the function call sequences for both the initialization and termination tasks.

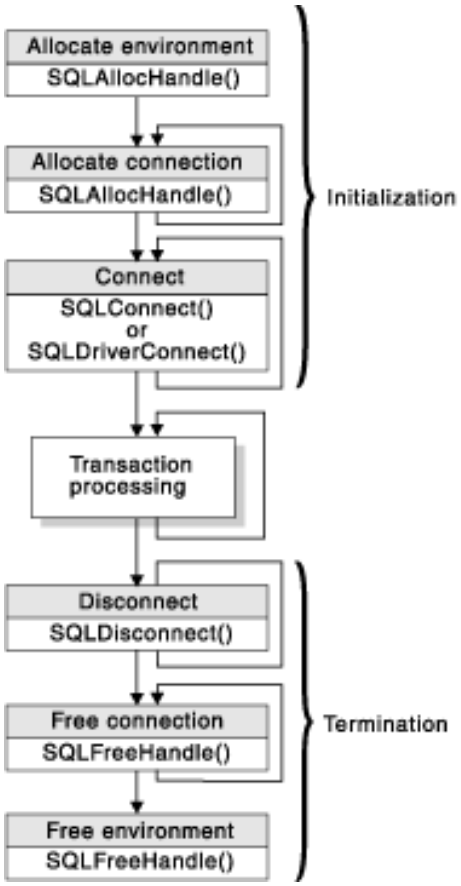



*Figure 2. Conceptual view of initialization and termination tasks*

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.

**Related concepts**

Transaction processing in Db2 ODBC
Transaction processing is the second core task after initialization. During this task, SQL statements query and modify data in Db2 ODBC.

## Handles

A *handle* is a variable that refers to a data object that is controlled by Db2 ODBC. The environment, connection, and statement handles are necessary for the initialization and termination processes.

Using handles relieves the application from managing 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.

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

**Statement handles**
A statement handle refers to the data object that describes and tracks the execution of an SQL statement. You can allocate a statement handle by calling `SQLAllocHandle()` and must do so before you can execute a statement.

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.

**Related concepts**

Statement handle allocation
A *statement handle* refers to the data object that describes and tracks the execution of an SQL statement. You must allocate a statement handle before you can execute a statement.

How to connect to one or more data sources
Db2 ODBC supports different connection types to remote data sources through DRDA.

## ODBC connection model

The ODBC specifications support any number of concurrent connections, each of which is an independent transaction.

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**
Db2 ODBC does not fully support the ODBC connection model if the initialization file does not specify MULTICONTEXT=1.

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.

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 roll back 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.

**Related concepts**

How to specify the connection type
Every IBM RDBMS supports both type 1 and type 2 connection type semantics, in which only one transaction is active at any time.

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.

**Related reference**

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

## How to specify the connection type

Every IBM RDBMS supports both type 1 and type 2 connection type semantics, in which 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.
- 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)).

**Related concepts**

How to use the initialization file

The Db2 ODBC initialization file is read at application run time. You can specify the file by using either a DSNAOINI data definition statement or by defining a DSNAOINIz/OS UNIX environment variable.

## How to connect to one or more data sources

Db2 ODBC supports different connection types 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. However, statements that execute on different connections do not coordinate.

**Example:** The following example 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]);
    /*********   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:
        return (SQL_ERROR);
    } else {
        printf(">Connected to
        return (SQL_SUCCESS);
    }
}
/* ... */
```

*Figure 3. An application that connects to two data sources*

**Related concepts**

ODBC and distributed units of work

You can write Db2 ODBC applications to use distributed units of work.

## Transaction processing in Db2 ODBC

Transaction processing is the second core task after initialization. During this task, SQL statements query and modify data in Db2 ODBC.

The following figure shows the typical order of function calls in a Db2 ODBC application. It does not show all functions or possible paths.

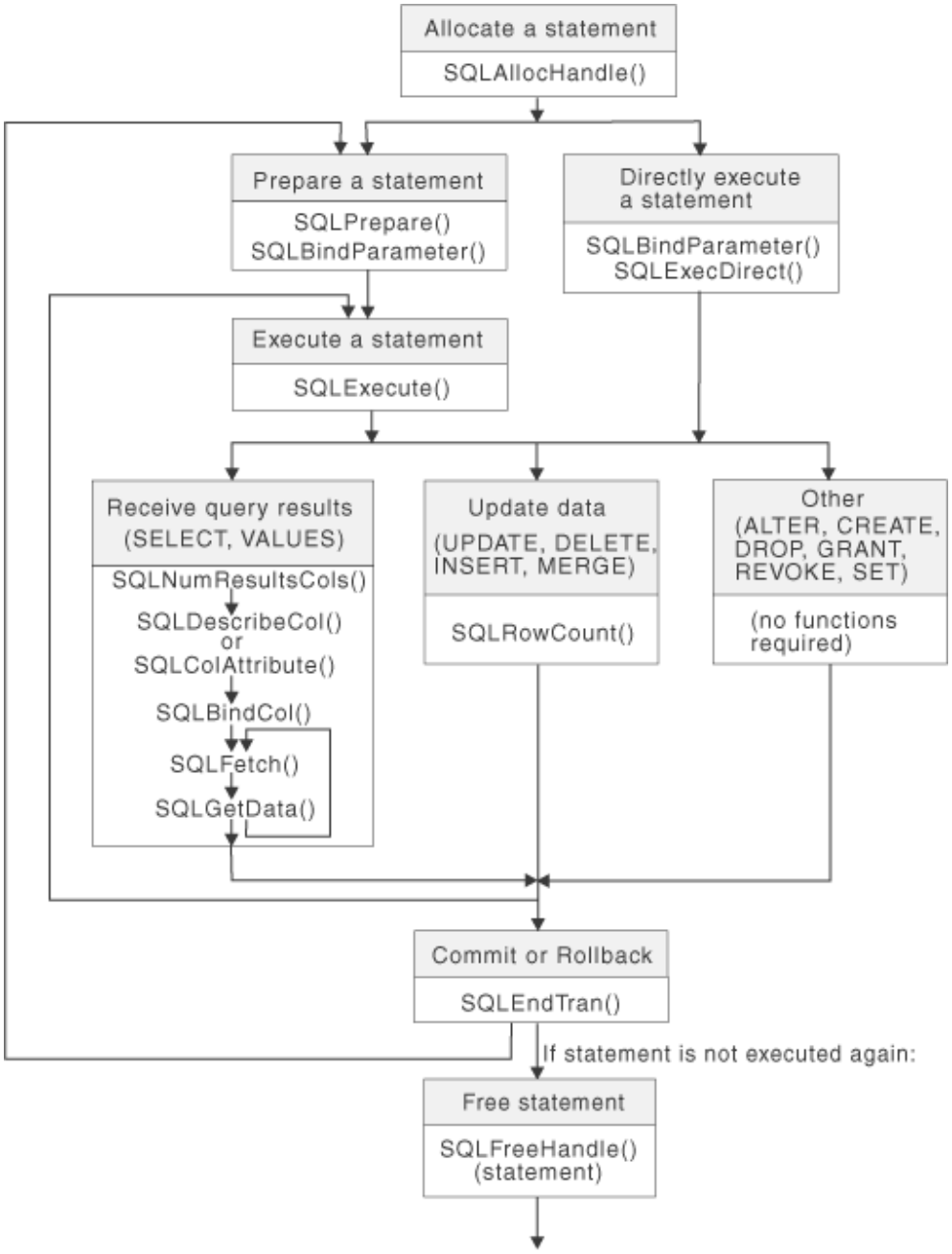



*Figure 4. Transaction processing*

The transaction processing 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

## Statement handle allocation

A *statement handle* refers to the data object that describes and tracks the execution of an SQL statement. You must allocate a statement handle before you can execute a statement.

`SQLAllocHandle()` (with *HandleType* set to SQL_HANDLE_STMT) allocates a statement handle to describe 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.

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 on page 16 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 of SQL statements

After you allocate a statement handle, you can specify and execute SQL statements.

You can execute SQL statements through the following steps:

- Prepare then execute:
  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:
  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 for z/OS and Db2 for Linux, UNIX, and Windows 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.

**Functions for binding parameters in SQL statements**
Both `SQLPrepare()`, followed by `SQLExecute()`, and `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.

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 database server or an error can occur.

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.

**Related concepts**
Input and retrieval of long data in pieces
When an application must manipulate long data values, loading the entire values into storage can become impractical. For this reason, Db2 ODBC provides a technique that enables you to handle long data values in pieces.

Data types and data conversion
When you write a Db2 ODBC application, you must work with both SQL data types and C data types. The database server uses SQL data types, and the application uses C data types.

Cast parameter markers to distinct types or distinct types to source types

When you use a distinct-type parameter in the predicate of a query statement, you must use a CAST function. You can cast either the parameter marker to a distinct type, or you can cast the distinct type to a source type.

**Related reference**

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

`SQLBindParameter()` binds parameter markers to application variables and extends the capability of the `SQLSetParam()` function.

## How an ODBC program processes results

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

### Processing query (SELECT, VALUES) statements

While processing query statements, the application must also run diagnostic checks.

**About this task**

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.

**Procedure**

To process query statements in an ODBC application:

1. Analyze the executed or prepared statement and describe the structure of the result set, including the number, types, and lengths of the columns.

   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.
2. Optional: To bind application variables to columns in order to receive the data, retrieve column data directly into an application variable on the next call to `SQLFetch()`.

   For each column to be retrieved, the application calls `SQLBindCol()` to bind an application variable to a column in the result set. The application 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.
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.

If data conversion was indicated by the data types specified on the call to SQLBindCol(), the conversion occurs when SQLFetch() is called.

4. Optional: Call SQLGetData() to retrieve columns that were not previously bound.

   Data conversion can also be indicated here, as in SQLBindCol(), by specifying the target C data type of the application variable.

   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.

**Related concepts**

Input and retrieval of long data in pieces
When an application must manipulate long data values, loading the entire values into storage can become impractical. For this reason, Db2 ODBC provides a technique that enables you to handle long data values in pieces.

Data types and data conversion
When you write a Db2 ODBC application, you must work with both SQL data types and C data types. The database server uses SQL data types, and the application uses C data types.

Retrieval of a result set into an array
An application can issue a query statement and fetch rows from the result set that the query generates.

Functions for setting and retrieving environment, connection, and statement attributes
Db2 ODBC provides functions that let you set or retrieve a subset of environment, connection, and statement attributes.

**Related reference**

SQLBindCol() - Bind a column to an application variable
SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLColAttribute() - Get column attributes
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.

SQLDescribeCol() - Describe column attributes
SQLDescribeCol() returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLGetData() - Get data from a column
SQLGetData() retrieves data for a single column in the current row of the result set. You can also use SQLGetData() to retrieve large data values in pieces. After you call SQLGetData() for each column, call SQLFetch() or SQLExtendedFetch() for each row that you want to retrieve.

SQLNumResultCols() - Get number of result columns
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 you call

SQLNumResultCols(). After you call SQLNumResultCols(), you can call SQLColAttribute() or one of the bind column functions.

**Processing UPDATE, DELETE, INSERT, and MERGE statements**
In some cases, you might need to use a cursor if you perform a positioned UPDATE or DELETE in your application. Otherwise, you need to check only for diagnostic messages.

**About this task**

If a statement modifies data (UPDATE, DELETE, INSERT, or MERGE 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.

**Procedure**

To perform a positioned update or delete in your application:

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 generated in the preceding step.

   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” on page 20.

```
sprintf((char *)stmtPositionedUpdate,
"UPDATE org SET location = 'San Jose'  WHERE CURRENT of %s",
 cursorName);
```

6. Execute the positioned update or delete statement.

**Related concepts**
Positioned updates of columns (Db2 SQL)
**Related tasks**
Cursors (Db2 Application programming and SQL)
**Related reference**
SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLGetCursorName() - Get cursor name
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.

SQLRowCount() - Get row count
SQLRowCount() returns the number of rows in a table that were affected by an UPDATE, INSERT, DELETE, or MERGE statement. You can call SQLRowCount() against a table or against a view that is based on the table. SQLExecute() or SQLExecDirect() must be called before SQLRowCount() is called.

SQLSetCursorName() - Set cursor name
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.

**Processing other statements**
You do not need to take further action other than a normal check for diagnostic messages if the statement neither queries nor modifies data.

## Commit and rollback in Db2 ODBC

Db2 ODBC supports two commit modes: autocommit and manual-commit. 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.

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. 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. An application can switch between manual-commit and autocommit modes by calling SQLSetConnectAttr(). Typically, a query-only application might want 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, transactions on each connection do not coordinate.

**Related concepts**
ODBC and distributed units of work
You can write Db2 ODBC applications to use distributed units of work.

Use of ODBC for querying the Db2 catalog
You can use Db2 ODBC catalog query functions and direct catalog queries to the Db2 ODBC shadow catalog to obtain catalog information.

**When to call SQLEndTran()**
In manual-commit mode, SQLEndTran() must be called before SQLDisconnect() is called.

An application in autocommit mode is not required to call SQLEndTran() because a commit is issued implicitly at the end of each statement execution. If distributed unit of work is involved, additional rules can apply.

**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 starting the `SQLSetConnectAttr()` function.

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.

**Related concepts**

ODBC and distributed units of work
You can write Db2 ODBC applications to use distributed units of work.

Functions for setting and retrieving environment, connection, and statement attributes
Db2 ODBC provides functions that let you set or retrieve a subset of environment, connection, and statement attributes.

**Related reference**

SQLSetConnectAttr() - Set connection attributes
`SQLSetConnectAttr()` sets attributes that govern aspects of connections.

**Effects of calling `SQLEndTran()`**
When a transaction ends, an application behaves with certain characteristics.

- All locks on database server 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 (as Db2 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).

**Related reference**

SQLSetStmtOption() - Set statement attribute
This function is deprecated and is replaced by SQLSetStmtAttr(). You cannot use `SQLSetStmtOption()` for 64-bit applications.

SQLTransact() - Transaction management
`SQLTransact()` is a deprecated function and is replaced by `SQLEndTran()`.

## Function for freeing statement handles

You call the `SQLFreeHandle()` function (with *HandleType* set to SQL_HANDLE_STMT) to terminate processing for a particular statement handle.

`SQLFreeHandle()` 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 must 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 deal with warning or error conditions that are generated within an application.

Db2 ODBC functions generate 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.

**Related concepts**

Problem diagnosis
Several guidelines exist for working with the Db2 ODBC traces, including information about general diagnosis, debugging, and abnormal terminations.

**Related reference**

SQLGetDiagRec() - Get multiple field settings of diagnostic record
`SQLGetDiagRec()` returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information. `SQLGetDiagRec()` also returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the error message text.

## Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

The following table 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**'. |

*Table 3. Db2 ODBC function return codes (continued)*

| Return code | Explanation |
|---|---|
| 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.. |
| 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. |

**Related concepts**

Input and retrieval of long data in pieces

When an application must manipulate long data values, loading the entire values into storage can become impractical. For this reason, Db2 ODBC provides a technique that enables you to handle long data values in pieces.

**Related reference**

SQLSTATE cross reference

SQLSTATEs are returned by the Db2 ODBC application as a diagnostic tool for encountered errors, indicating the cause for these errors.

## SQLSTATEs for ODBC error reporting

Db2 ODBC provides a standard set of codes called *SQLSTATEs* because different database servers often have different diagnostic message codes. Certain guidelines apply to the use of SQLSTATEs within applications.

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 *ccsss*, where *cc* indicates class and *sss* indicates subclass. All SQLSTATEs use one of the following classes:

**'01'**
A warning.

**'S1'**
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 ISO CLI standard.

For some error conditions, Db2 ODBC returns SQLSTATEs that differ from those states listed in Microsoft open database connectivity (ODBC). 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 refer to the application that makes 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.

**Related reference**

SQLSTATE cross reference
SQLSTATEs are returned by the Db2 ODBC application as a diagnostic tool for encountered errors, indicating the cause for these errors.

SQLSTATE mappings
Several SQLSTATEs differ when you call SQLGetDiagRec() or SQLError() under an ODBC 3.0 driver. All deprecated functions continue to return ODBC 2.0 SQLSTATEs regardless of which environment attributes are set.

### SQLCA retrieval in an ODBC application

Embedded applications rely on the SQLCA data structure for all diagnostic information. The SQLGetSQLCA() function is used to retrieve this data structure.

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).

**Related reference**

SQLGetSQLCA() - Get SQLCA data structure
SQLGetSQLCA() returns the SQLCA (SQL communication area) that is associated with preparing and executing an SQL statement, fetching data, or closing a cursor. The SQLCA can return supplemental information about the data that is obtained by SQLGetDiagRec().

## Data types and data conversion

When you write a Db2 ODBC application, you must work with both SQL data types and C data types. The database server uses SQL data types, and the application uses C data types.

The application must therefore match C data types to SQL data types when transferring data between the database server 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 database server 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

Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

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 on page 26 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 database server.

**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.

**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 28. 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 |
|---|---|---|
| BIGINT | SQL_BIGINT | SQL_C_BIGINT |
| BINARY | SQL_BINARY | SQL_C_BINARY |
| BLOB | SQL_BLOB | SQL_C_BINARY |
| BLOB LOCATOR"1" on page 27 | SQL_BLOB_LOCATOR | SQL_C_BLOB_LOCATOR |
| CHAR | SQL_CHAR | SQL_C_CHAR |
| CHAR FOR BIT DATA "6" on page 27 | 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" on page 27 | SQL_C_TYPE_DATE |
| DBCLOB | SQL_DBCLOB | SQL_C_DBCHAR |
| DBCLOB LOCATOR"1" on page 27 | SQL_DBCLOB_LOCATOR | SQL_C_DBCLOB_LOCATOR |
| DECFLOAT(16) or DECFLOAT(34) | SQL_DECFLOAT | SQL_C_CHAR |
| 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"4" on page 27 |
| INTEGER | SQL_INTEGER | SQL_C_LONG |
| LONG VARCHAR"5" on page 27 | SQL_LONGVARCHAR | SQL_C_CHAR |
| LONG VARCHAR FOR BIT DATA"5" on page 27,"6" on page 27 | SQL_LONGVARBINARY | SQL_C_BINARY |

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

| SQL data type | Symbolic SQL data type | Default symbolic C data type |
|---|---|---|
| LONG VARGRAPHIC"5" on page 27 | SQL_LONGVARGRAPHIC | SQL_C_DBCHAR or SQL_C_WCHAR"4" on page 27 |
| NUMERIC"7" on page 27 | SQL_NUMERIC"7" on page 27 | SQL_C_CHAR |
| REAL | SQL_REAL | SQL_C_FLOAT |
| ROWID | SQL_ROWID | SQL_C_CHAR |
| SMALLINT | SQL_SMALLINT | SQL_C_SHORT |
| TIME | SQL_TYPE_TIME"2" on page 27 | SQL_C_TYPE_TIME |
| TIMESTAMP | SQL_TYPE_TIMESTAMP"2" on page 27 | SQL_C_TYPE_TIMESTAMP SQL_C_TYPE_TIMESTAMP_EXT"3" on page 27 |
| VARBINARY | SQL_VARBINARY | SQL_C_BINARY |
| VARCHAR | SQL_VARCHAR | SQL_C_CHAR |
| VARCHAR FOR BIT DATA"6" on page 27 | SQL_VARBINARY | SQL_C_BINARY |
| VARGRAPHIC | SQL_VARGRAPHIC | SQL_C_DBCHAR or SQL_C_WCHAR"4" on page 27 |
| XML | SQL_XML | SQL_C_BINARY |

**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).
2. Changes to datetime data types have been made since previous releases.
3. SQL_C_TYPE_TIMESTAMP_EXT is used for TIMESTAMP values in which the fractional precision is 0 to 12 digits.
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.
5. Whenever possible, replace LONG data types with LOB types.
6. Whenever possible, replace FOR BIT DATA data types with BINARY or VARBINARY types.
7. NUMERIC is a synonym for DECIMAL on Db2 for z/OS, DB2 for VSE & VM and Db2 for Linux, UNIX, and Windows.

**Additional information:**

- 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 28 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.

**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.

**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 29.

*Table 5. C data types*

| C symbolic data type | C type | Base C type |
|---|---|---|
| SQL_C_BIGINT | SQLBIGINT | long long int |
| 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 (31-bit) or int (64-bit)"1" on page 29 |
| SQL_C_DOUBLE | SQLDOUBLE | Double |
| SQL_C_FLOAT | SQLREAL | Float |
| SQL_C_DECIMAL64 | SQLDECIMAL64 | See Table 6 on page 29 |
| SQL_C_DECIMAL128 | SQLDECIMAL128 | See Table 6 on page 29 |
| SQL_C_TYPE_DATE"2" on page 29 | DATE_STRUCT | See Table 6 on page 29 |
| SQL_C_TYPE_TIMESTAMP"2" on page 29 | TIMESTAMP_STRUCT | See Table 6 on page 29 |
| SQL_C_TYPE_TIME-STAMP_EXT "2" on page 29 | TIMESTAMP_STRUCT_EXT | See Table 6 on page 29 |
| SQL_C_CLOB_LOCATOR | SQLINTEGER | Long int (31-bit) or int (64-bit)"1" on page 29 |
| SQL_C_BINARY | SQLCHAR | Unsigned char |
| SQL_C_BINARYXML | SQLCHAR | Unsigned char |
| SQL_C_BLOB_LOCATOR | SQLINTEGER | Long int (31-bit) or int (64-bit)"1" on page 29 |
| SQL_C_DBCHAR | SQLDBCHAR | Unsigned short |
| SQL_C_DBCLOB_LOCATOR | SQLINTEGER | Long int (31-bit) or int (64-bit)"1" on page 29 |
| SQL_C_WCHAR | SQLWCHAR | wchar_t (31-bit) or unsigned short (64-bit)"1" on page 29 |

*Table 5. C data types (continued)*

| C symbolic data type | C type | Base C type |
|---|---|---|

**Note:**

1. 31-bit is for 31-bit applications, and 64-bit is for 64 bit applications.

   In the 31-bit environment, long int is 32 bits. In the 64-bit environment, int is also 32 bits. Therefore, the C type SQLINTEGER is mapped to a 32-bit field regardless of the environment.

   In the 31-bit environment, wchar_t is 16 bits. In the 64-bit environment, unsigned short is also 16 bits. Therefore, the C type SQLWCHAR is mapped to a 16-bit field regardless of the environment.
2. Changes to datetime data types have been made since previous releases.

The following table lists the C data types with their associated structures for date, time, timestamp, and decimal floating point.

*Table 6. C date, time, timestamp, and decimal floating point structures*

| C type | Generic structure |
|---|---|
| DATE_STRUCT | `typedef struct DATE_STRUCT`<br>`  {`<br>`  SQLSMALLINT  year;`<br>`  SQLUSMALLINT  month;`<br>`  SQLUSMALLINT  day;`<br>`  } DATE_STRUCT;` |
| TIME_STRUCT | `typedef struct TIME_STRUCT`<br>`  {`<br>`  SQLUSMALLINT  hour;`<br>`  SQLUSMALLINT  minute;`<br>`  SQLUSMALLINT  second;`<br>`  } TIME_STRUCT;` |
| TIMESTAMP_STRUCT | `typedef struct TIMESTAMP_STRUCT`<br>`  {`<br>`  SQLUSMALLINT  year;`<br>`  SQLUSMALLINT  month;`<br>`  SQLUSMALLINT  day;`<br>`  SQLUSMALLINT  hour;`<br>`  SQLUSMALLINT  minute;`<br>`  SQLUSMALLINT  second;`<br>`  SQLINTEGER  fraction;`<br>`  } TIMESTAMP_STRUCT;` |
| TIME-<br>STAMP_STRUCT_EXT | `typedef struct TIMESTAMP_STRUCT_EXT`<br>`{`<br>`  SQLUSMALLINT year;`<br>`  SQLUSMALLINT month;`<br>`  SQLUSMALLINT day;`<br>`  SQLUSMALLINT hour;`<br>`  SQLUSMALLINT minute;`<br>`  SQLUSMALLINT second;`<br>`  SQLINTEGER fraction;`<br>`  SQLINTEGER fraction2;`<br>`} TIMESTAMP_STRUCT_EXT;` |
| SQLDECIMAL64 | `typedef struct SQLDECIMAL64`<br>`{`<br>`  union{`<br>`  SQLDOUBLE dummy;`<br>`  SQLCHAR dec64[8];`<br>`  }dec64;`<br>`} SQLDECIMAL64;` |

*Table 6. C date, time, timestamp, and decimal floating point structures (continued)*

| C type | Generic structure |
|---|---|
| SQLDECIMAL128 | `typedef struct SQLDECIMAL128`<br>`{`<br>`  union{`<br>`  SQLDOUBLE dummy;`<br>`  SQLCHAR dec128[16];`<br>`  }dec128;`<br>`} SQLDECIMAL128;` |

**Related concepts**

Using LOBs

The term large object (LOB) refers to any type of large object. Db2 supports three LOB data types: binary large object (BLOB), character large object (CLOB), and double-byte character large object (DBCLOB).

Application encoding schemes and Db2 ODBC

Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The Db2 ODBC driver supports input and output character string arguments to ODBC APIs and input and output host variable data in each of these encoding schemes.

**Related reference**

Changes to datetime data types

The ODBC driver supports both ODBC 2.0 and ODBC 3.0 datetime values for input. For output, the ODBC driver determines the value to return based on the setting of the environment attribute.

SQLBindCol() - Bind a column to an application variable

`SQLBindCol()` binds a column to an application variable. You can call `SQLBindCol()` once for each column in a result set from which you want to retrieve data or LOB locators.

SQLDescribeCol() - Describe column attributes

`SQLDescribeCol()` returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either `SQLPrepare()` or `SQLExecDirect()`.

C data types that do not map to SQL 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.

## C data types that do not map to SQL 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.

The following table 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 (31-bit) or int (64-bit)[1] | Handle referencing environment information. |
| SQLHDBC | long int (31-bit) or int (64-bit)[1] | Handle referencing data source connection information. |
| SQLHSTMT | long int (31-bit) or int (64-bit)[1] | Handle referencing statement information. |
| SQLUSMALLINT | unsigned short int | Function input argument for unsigned short integer values. |

*Table 7. C data types and base C data types (continued)*

| Defined C type | Base C type | Typical usage |
|---|---|---|
| SQLUINTEGER | unsigned long int (31-bit) or unsigned int (64-bit)[1] | Function input argument for unsigned long integer values. |
| SQLLEN | int | Function input or output argument for 32-bit integer values. |
| SQLULEN | unsigned int | Function input or output argument for unsigned 32-bit integer values. |
| SQLRETURN | short int | Return code from Db2 ODBC functions. |
| SQLWCHAR | wchar_t (31-bit) or unsigned short (64-bit)[1] | Data type for a Unicode UCS-2 character. |
| SQLWCHAR * | wchar_t * (31-bit) or unsigned short * (64-bit)[1] | Pointer to storage for Unicode UCS-2 data. |

**Note:**

1. 31-bit is for 31-bit applications, and 64-bit is for 64 bit applications.

   In the 31-bit environment, long int and unsigned long int are each 32 bits. In the 64-bit environment, int and unsigned int are also each 32 bits. Therefore, the C types SQLHENV, SQLHDBC, SQLHSTMT, and SQLUINTEGER are each mapped to 32-bit fields regardless of the environment.

   In the 31-bit environment, wchar_t is 16 bits. In the 64-bit environment, unsigned short is also 16 bits. Therefore, the C type SQLWCHAR is mapped to a 16-bit field regardless of the environment.

   Pointers do not have the same size in a 64-bit environment as they do in a 31-bit environment. In the 31-bit environment, SQLWCHAR * is 32 bits long, and in the 64-bit environment, SQLWCHAR * is 64 bits long.

## Data conversion

Db2 ODBC manages the transfer and any required conversion of data between the application and the database server. However, not all data conversions are supported.

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 symbolic type names shown 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);
```

Supported data conversions lists the data conversions that Db2 ODBC supports.

The first column contains the data type of the SQL data type. The remaining columns represent the C data types. If the C data type columns contain:

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

**X**
All IBM databases support the conversion.

**blank**
No IBM databases supports the conversion.

*Table 8. Supported data conversions*

| SQL data type | SQL_C_CHAR | SQL_C_WCHAR | SQL_C_BINARYXML | SQL_C_DBCHAR | SQL_C_BIT | SQL_C_TINYINT | SQL_C_SHORT | SQL_C_LONG | SQL_C_BIGINT | SQL_C_SBIGINT | SQL_C_FLOAT | SQL_C_DOUBLE | SQL_C_DECIMAL64 | SQL_C_DECIMAL128 | SQL_C_TYPE_DATE | SQL_C_TYPE_TIME | SQL_C_TYPE_TIMESTAMP | SQL_C_TYPE_TIMESTAMP_EXT | SQL_C_TYPE_TIMESTAMP_EXT_TZ | SQL_C_BINARY | SQL_C_CLOB_LOCATOR | SQL_C_BLOB_LOCATOR | SQL_C_DBCLOB_LOCATOR |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQL_CHAR | D[1] | X[2] | | | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | | | |
| SQL_ VARCHAR | D[1] | X[2] | | | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | | | |
| SQL_ LONG VARCHAR | D[1] | X[2] | | | | | | | | | | | X | X | | | | | | X | | | |
| SQL_ BINARY | X[1] | X[2] | | | | | | | | | | | | | | | | | | D | | | |
| SQL_ VARBINARY | X[1] | X[2] | | | | | | | | | | | | | | | | | | D | | | |
| SQL_ LONG VARBINARY | X[1] | X[2] | | | | | | | | | | | | | | | | | | D | | | |
| SQL_ GRAPHIC[4] | X[1] | D[2] | | D | | | | | | | | | | | | | | | | | | | |
| SQL_ VARGRAPHIC[4] | X[1] | D[2] | | D | | | | | | | | | | | | | | | | | | | |
| SQL_ LONG VARGRAPHIC[4] | X[1] | D[2] | | D | | | | | | | | | | | | | | | | | | | |
| SQL_CLOB | D[1] | X[2] | | | | | | | | | | | | | | | | | | X | X[3] | | |
| SQL_BLOB | X[1] | X[2] | | | | | | | | | | | | | | | | | | D | | X[3] | |
| SQL_DBCLOB | X[1] | X[2] | | D | | | | | | | | | | | | | | | | | | | X[3] |
| SQL_ CLOB_ LOCATOR | X[1] | X[2] | | | | | | | | | | | | | | | | | | X | D[3] | | |
| SQL_ BLOB_ LOCATOR | X[1] | X[2] | | | | | | | | | | | | | | | | | | X | | D[3] | |
| SQL_ DBCLOB_ LOCATOR | X[1] | X[2] | | X | | | | | | | | | | | | | | | | | | | D[3] |
| SQL_NUMERIC[5] | D[1] | X[2] | | | X | X | X | X | X | X | X | X | X | X | | | | | | X | | | |
| SQL_DECIMAL | D[1] | X[2] | | | X | X | X | X | X | X | X | X | X | X | | | | | | X | | | |
| SQL_DECFLOAT | D[1] | X[2] | | | X | X | X | X | X | X | X | X | X | X | | | | | | | | | |
| SQL_INTEGER | X[1] | X[2] | | X | X | X | X | D | X | X | X | X | X | X | | | | | | X | | | |
| SQL_ SMALLINT | X[1] | X[2] | | | X | X | D | X | X | X | X | X | X | X | | | | | | X | | | |
| SQL_FLOAT | X[1] | X[2] | | | X | X | X | X | X | X | X | D | X | X | | | | | | X | | | |
| SQL_DOUBLE | X[1] | X[2] | | | X | X | X | X | X | X | X | D | X | X | | | | | | X | | | |
| SQL_REAL | X[1] | X[2] | | | X | X | X | X | X | X | D | X | X | X | | | | | | X | | | |
| SQL_BIGINT | X[1] | X[2] | | | X | X | X | X | D | D | X | X | X | X | | | | | | | | | |

*Table 8. Supported data conversions (continued)*

| SQL data type | SQL_C_CHAR | SQL_C_WCHAR | SQL_C_BINARYXML | SQL_C_DBCHAR | SQL_C_BIT | SQL_C_TINYINT | SQL_C_SHORT | SQL_C_LONG | SQL_C_BIGINT | SQL_C_SBIGINT | SQL_C_FLOAT | SQL_C_DOUBLE | SQL_C_DECIMAL64 | SQL_C_DECIMAL128 | SQL_C_TYPE_DATE | SQL_C_TYPE_TIME | SQL_C_TYPE_TIMESTAMP | SQL_C_TYPE_TIMESTAMP_EXT | SQL_C_TYPE_TIMESTAMP_EXT_TZ | SQL_C_BINARY | SQL_C_CLOB_LOCATOR | SQL_C_BLOB_LOCATOR | SQL_C_DBCLOB_LOCATOR |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQL_TYPE_DATE | X[1] | X[2] | | | | | | | | | | | | | D | | X | X | X | X | | | |
| SQL_TYPE_TIME | X[1] | X[2] | | | | | | | | | | | | | | D | X | X | X | X | | | |
| SQL_TYPE_TIMESTAMP | X[1] | X[2] | | | | | | | | | | | | | X | X | D | D | X | X | | | |
| SQL_TYPE_TIMESTAMP_WITH_ZONE | X[1] | X[2] | | | | | | | | | | | | | X | X | X | X | D | | | | |
| SQL_ROWID | D[1] | | | | | | | | | | | | | | | | | | | | | | |
| SQL_XML | X[1] | X[2] | X | X | | | | | | | | | | | | | | | | D | | | |

**Note:**

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.
5. NUMERIC is a synonym for DECIMAL on Db2 for z/OS, DB2 Server for VSE & VM, and Db2 for Linux, UNIX, and Windows.

Limits on precision, and scale, as well as truncation and rounding rules are the same as those for Db2 for z/OS, 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.

**Related concepts**

Application encoding schemes and Db2 ODBC

Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The Db2 ODBC driver supports input and output character string arguments to ODBC APIs and input and output host variable data in each of these encoding schemes.

**Related reference**

C and SQL data types

Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

Data conversion between the application and the database server
Data conversion is possible between C and SQL data types. You need to know the precision, scale, length, and display size of each data type. In addition, you will also need to know how to convert from one data type to the other.

Limits in Db2 for z/OS (Db2 SQL)

# Characteristics of string arguments

String arguments in Db2 ODBC rely on conventions.

## Length of string arguments

String arguments for both output and input have associated length arguments. You should always use a valid output length argument.

For input string arguments, the associated length 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, character strings that Db2 ODBC return are terminated with a nul-terminator (hex 00), except for strings that are 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 the buffer, Db2 ODBC truncates the string to the size of the buffer and writes the nul-terminator. The string data can be truncated on input or output by using the appropriate SQLSTATE.

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

Db2 ODBC normally interprets string arguments in a case-sensitive manner and does not trim any spaces from the values.

An 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.

## Functions for querying environment and data source information

Db2 ODBC provides functions that let applications retrieve information about the characteristics and capabilities of the current ODBC driver or the data source to which it is connected.

One of the most common situations in which functions are needed that provide driver or data source information 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.

**Example:** The following code 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 ZOSDB2 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, "ZOSDB2" ) )   /* data source exist          */
       {
           found = TRUE;
           break;
       }
    }
    if( !found )
    {
        fprintf(stdout, "Data source
        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
    }
    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*

**Related reference**

SQLDataSources() - Get a list of data sources
`SQLDataSources()` returns a list of available target databases, one at a time. Before you make a connection, you can call `SQLDataSources()` to determine which databases are available.

SQLGetFunctions() - Get functions
`SQLGetFunctions()` indicates if a specific function is supported.

SQLGetInfo() - Get general information
`SQLGetInfo()` returns general information about the database management systems to which the application is currently connected. For example, `SQLGetInfo()` indicates which data conversions are supported.

SQLGetTypeInfo() - Get data type information
`SQLGetTypeInfo()` returns information about the data types that are supported by the database management systems that are associated with Db2 ODBC. This information is returned in an SQL result set. The columns of this result set can be received by using the same functions that you use to process a query.

# Chapter 3. Configuring Db2 ODBC and running sample applications

Before you prepare and run ODBC applications, you need to install Db2 ODBC and run the sample applications.

## Running the SMP/E jobs for Db2 ODBC installation

To install Db2 ODBC, you must edit and run SMP/E jobs.

**Procedure**

To run the SMP/E jobs for Db2 ODBC installation:

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.
5. 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.

**Related tasks**

Editing the SMP/E jobs (Db2 Installation and Migration)

## The Db2 ODBC run time environment

Db2 ODBC support is implemented as an IBM C/C++ Dynamic Load Library (DLL). All API calls are routed through the single ODBC driver that is loaded at run time into the application address space.

Because Db2 ODBC support is provided as 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 for z/OS address space.

- If the Db2 ODBC application is **not** running as a Db2 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 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.

The following versions of the ODBC driver are available on Db2 for z/OS:

- the 31-bit XPLINK driver
- the 31-bit non-XPLINK driver
- the 64-bit driver

You specify which driver your application uses by which definition sidedeck you include when you prelink and link-edit your application.

Db2 ODBC supports access to the local Db2 for z/OS subsystems and any remote data source that is accessible using Db2 for z/OS Db2 11. This includes:

© Copyright IBM Corp. 1997, 2020

- 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 for z/OS ODBC driver and the Db2 for z/OS subsystem are illustrated in the following figure.

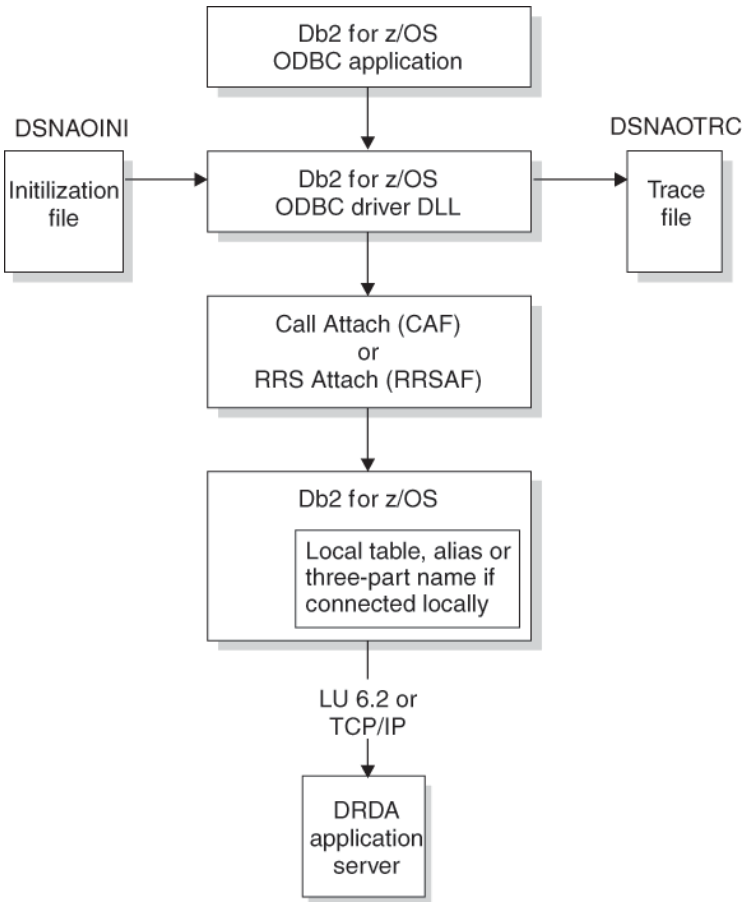



*Figure 6. Relationship between Db2 for z/OS ODBC components*

**Related tasks**

Prelinking and link-editing an ODBC application

After you compile your ODBC application, you must prelink and link-edit it before you can run it. This process is slightly different depending on whether the application is an XPLINK application, a non-XPLINK application, or a 64-bit application.

## Connectivity requirements

You must run Db2 ODBC applications on a machine that has Db2 11 for z/OS installed. Additional requirements also apply.

Db2 for z/OS ODBC has the following additional connectivity requirements:

- 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) 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.

**Related concepts**

How to specify the connection type

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

### Extra performance linkage

The XPLINK Db2 ODBC driver enhances the performance of XPLINK ODBC applications. The XPLINK Db2 ODBC driver is recommended to enhance performance only if your ODBC application uses XPLINK code exclusively.

If you use any non-XPLINK code in your application, the XPLINK ODBC driver might not increase performance. The non-XPLINK ODBC driver is recommended for applications that include non-XPLINK code.

**Related concepts**

Overview of preparing and executing a Db2 ODBC application
To prepare and execute a Db2 ODBC application, you need to follow certain steps and understand the Db2 ODBC components.

**Related tasks**

Compiling 31-bit XPLINK applications
If your 31-bit application needs to use z/OS XPLINK function linkage, use the ODBC 31-bit XPLINK driver and compile your application as an XPLINK application.

Executing an ODBC application
You can execute an ODBC application by using a z/OS job or by using z/OS UNIX commands.

Link-editing 31-bit XPLINK applications
For applications that are compiled as 31-bit XPLINK applications, you must link-edit your applications with the DFSMS binder.

### 64-bit ODBC driver

For 64-bit applications, you must use the 64-bit ODBC driver. This driver enables your application to access storage above the 2-GB bar.

The 64-bit driver is an XPLINK driver. A non-XPLINK driver does not exist for 64-bit applications.

**Related concepts**

Extra performance linkage
The XPLINK Db2 ODBC driver enhances the performance of XPLINK ODBC applications. The XPLINK Db2 ODBC driver is recommended to enhance performance only if your ODBC application uses XPLINK code exclusively.

## Db2 ODBC run time environment setup

The steps in setting up the Db2 ODBC run time environment must be performed once. These steps are performed as part of the installation process for Db2 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 steps are the same regardless of the version of the ODBC driver (31-bit non-XPLINK, 31-bit XPLINK, or 64-bit) that you use.

The online bind sample is available in DSN1110.SDSNSAMP(DSNTIJCL). It is recommended that you use this bind sample as a guide for binding DBRMs to packages and binding an application plan.

## Binding DBRMs to create packages

You can bind database request modules (DBRMs) to create packages that use different isolation levels or default options.

### About this task

Use the online bind sample, DSN1110.SDSNSAMP(DSNTIJCL), for guidance.

Before an application can access data sources using Db2 ODBC, you must bind all required IBM DBRMs (which are shipped in DSN1110.SDSNDBRM) to all data sources. These data sources include the local Db2 for z/OS subsystem and all remote (DRDA) data sources.

To call stored procedures that run under Db2 ODBC, bind each of these procedures into packages at 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.

### Procedure

To bind the required DBRMs:

- Bind the following DBRMs to all data sources:
  - DSNCLINF
  - DSNCLICR

  You do not need to specify the ISOLATION bind option for DSNCLINF or DSNCLICR, because this option has no effect on the isolation level that the ODBC driver uses for the application. Instead, specify the isolation level in the ODBC initialization file with the keyword TXNISOLATION. The default value for this keyword is 2 for cursor stability (CS). Alternatively, you can use SQLSetConnectAttr() and SQLSetStmtAttr() to set the attribute SQL_ATTR_TXN_ISOLATION at the connection or statement level. This attribute overrides the keyword value.
- Bind the following DBRMs with default options to all 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 for i servers.
- Bind DSNCLIV1 and DSNCLIV2 with default options to all Db2 for Linux, UNIX, and Windows servers.
- Bind DSNCLIQR to any site that supports DRDA query result sets.

### Related concepts

Stored procedures for ODBC applications
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.

### Related tasks

Binding the application plan
When you bind the Db2 ODBC application plan, use the online bind sample, DSN1110.SDSNSAMP (DSNTIJCL), for guidance.

Migrating to the Db2 11 ODBC driver
When you migrate the ODBC driver for Db2 for z/OS from the Db2 10 driver to the Db2 11 driver, you must set up the Db2 11 ODBC run time environment.

### Related reference

BIND and REBIND options for packages, plans, and services (Db2 Commands)

SQLSetConnectAttr() - Set connection attributes
`SQLSetConnectAttr()` sets attributes that govern aspects of connections.

SQLSetStmtAttr() - Set statement attributes
`SQLSetStmtAttr()` sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call `SQLSetConnectAttr()`.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

**Impact of package bind options**
When you bind ODBC packages, you must specify certain values for several bind options. You should also consider specific ODBC recommendations for several other bind options.

The requirements and recommendations for the bind options are as follows:

- CURRENTDATA(NO)

  Binding the ODBC packages with CURRENTDATA(NO) reduces lock contention and processor utilization, which results in increased application concurrency and improved performance. Use of CURRENTDATA(NO) also allows block fetching for distributed, ambiguous cursors.
- DYNAMICRULES(BIND)

  Binding the ODBC 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.

  **Requirement:** You must specify ENCODING(EBCDIC) when you bind packages to the local Db2 for z/OS ODBC subsystem.
- SQLERROR(CONTINUE)

  **Important:** The SQLERROR(CONTINUE) bind option bypasses every error that occurs during the bind operation for the package. The recommendation is to ensure that only expected SQLCODEs are bypassed.

  Use SQLERROR(CONTINUE) for the following purposes:

  – When you bind DSNCLIMS on a down-level server. 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 try the bind again to bypass this error. Alternatively, you can bind with the SQLERROR(CONTINUE) keyword, however, the `SQLDescribeParam()` API will be unavailable to you at the down-level server.

  – When you bind DSNCLIMS on a Db2 subsystem with MIXED DATA=YES

Binding DSNCLIMS on any Db2 subsystem that is configured with MIXED DATA=YES results in SQLCODE -130. Bind with SQLERROR(CONTINUE) keyword to bypass this error.

– When you bind DSNCLIMS on a Db2 subsystem with MIXED DATA=NO

  Binding DSNCLIMS on any Db2 subsystem that is configured with MIXED DATA=NO results in SQLCODE -189. You can bind DSNCLIMS with SQLERROR(CONTINUE) to bypass this error. However, if you do this you cannot fetch from an ASCII DBCLOB column using the SQLGetData() API or LOB LOCATORs. Before you can do that, you must:

  - Define your Db2 subsystem with MIXED DATA=YES, with valid mixed and graphic ASCII CCSIDs.
  - Rebind DSNCLIMS.

**Related concepts**

Db2 ODBC run time environment setup
The steps in setting up the Db2 ODBC run time environment must be performed once. These steps are performed as part of the installation process for Db2 for z/OS.

**Related tasks**

Binding DBRMs to create packages
You can bind database request modules (DBRMs) to create packages that use different isolation levels or default options.

Binding the application plan
When you bind the Db2 ODBC application plan, use the online bind sample, DSN1110.SDSNSAMP (DSNTIJCL), for guidance.

Migrating to the Db2 11 ODBC driver
When you migrate the ODBC driver for Db2 for z/OS from the Db2 10 driver to the Db2 11 driver, you must set up the Db2 11 ODBC run time environment.

**Related reference**

BIND and REBIND options for packages, plans, and services (Db2 Commands)
MIXED DATA field (MIXED DECP value) (Db2 Installation and Migration)

**Return codes from ODBC package binding**

A bind to Db2 for z/OS produces several warning messages.

The expected warnings are listed as follows:

- For all packages:

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

- For bind of package DSNCLINC to Db2 for z/OS:

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

- For bind of package DSNCLIF4 to Db2 for z/OS for SYSIBM.LOCATIONS due to differences in catalog table names between releases.

## Binding the application plan

When you bind the Db2 ODBC application plan, use the online bind sample, DSN1110.SDSNSAMP (DSNTIJCL), for guidance.

**Procedure**

Use the PKLIST keyword to name all ODBC packages.

You can use any name for the plan. The default name is DSNACLI. If you select a name other than the default name, you must specify that name in the initialization file, by using the PLANNAME keyword.

Do not specify the DISCONNECT or CURRENTSERVER bind options when you bind the Db2 ODBC application plan. You must use the defaults.

Do not specify any PLAN bind options when you bind the application plan.

**Related tasks**

Binding DBRMs to create packages
You can bind database request modules (DBRMs) to create packages that use different isolation levels or default options.

## Setting up Db2 ODBC for the z/OS UNIX environment

You can compile, bind, and run ODBC applications in the z/OS UNIX environment. However, you must first set up the environment. You need to perform this setup task only once.

**Procedure**

Make the Db2 ODBC definition sidedeck available to z/OS UNIX users by performing one of the following actions:

- Define a data set alias for the ODBC sidedeck that uses .EXP as the last qualifier in the name. This alias must relate to the *prefix*.SDSNMACS data set where the Db2 ODBC definition sidedeck is installed.

  The z/OS UNIX environment compiler determines the contents of an input file based on the file extension. If a file resides 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 sidedeck if it meets these conditions:

  – It resides in a PDS.
  – The last qualifier in the PDS name is .EXP.

  For example, assume that Db2 is installed using DSN1110 as the high-level data set qualifier. You can define the alias by using the following command:

  ```
  DEFINE ALIAS(NAME('DSN1110.SDSNC.EXP') RELATE('DSN1110.SDSNMACS'))
  ```

  This alias allows z/OS UNIX environment users to directly reference the Db2 ODBC definition sidedeck by specifying the following input files as input to the z/OS UNIX environment c89 command.

  For the non-XPLINK ODBC driver:

  ```
  "//'DSN1110.SDSNC.EXP(DSNAOCLI)'"
  ```

  For the 31-bit XPLINK ODBC driver:

  ```
  "//'DSN1110.SDSNC.EXP(DSNAOCLIX)'"
  ```

  For the 64-bit XPLINK ODBC driver:

  ```
  "//'DSN1110.SDSNC.EXP(DSNAO64C)'"
  ```

- Specify the z/OS data set suffix by using the _C89_XSUFFIX_HOST or _CXX_XSUFFIX_HOST environment variable. 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"
  ```

# Overview of preparing and executing a Db2 ODBC application

To prepare and execute a Db2 ODBC application, you need to follow certain steps and understand the Db2 ODBC components.

The following figure shows the Db2 ODBC configuration process.

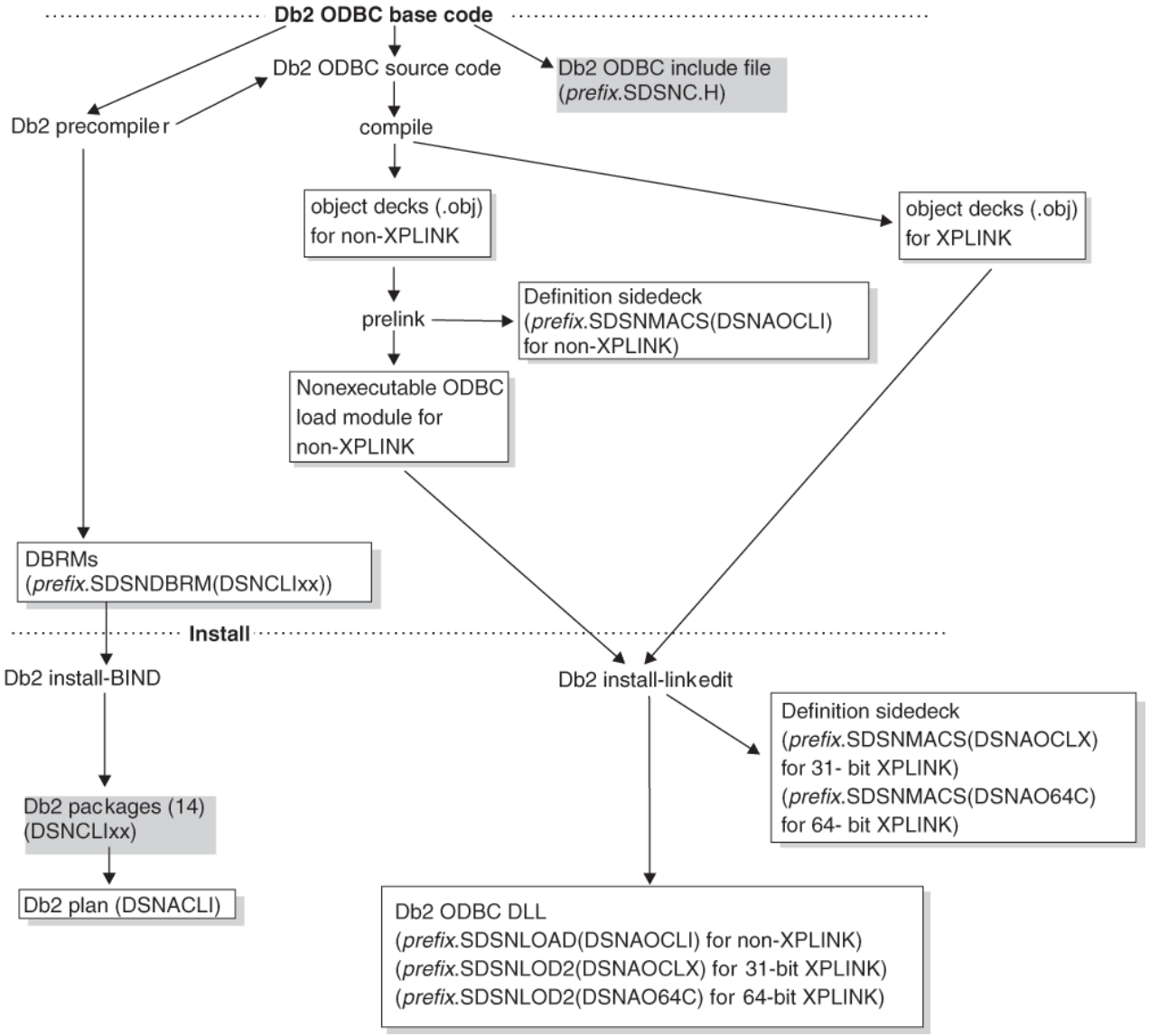



*Figure 7. Db2 ODBC customization*

The following figure shows the process you follow to prepare a Db2 ODBC application.

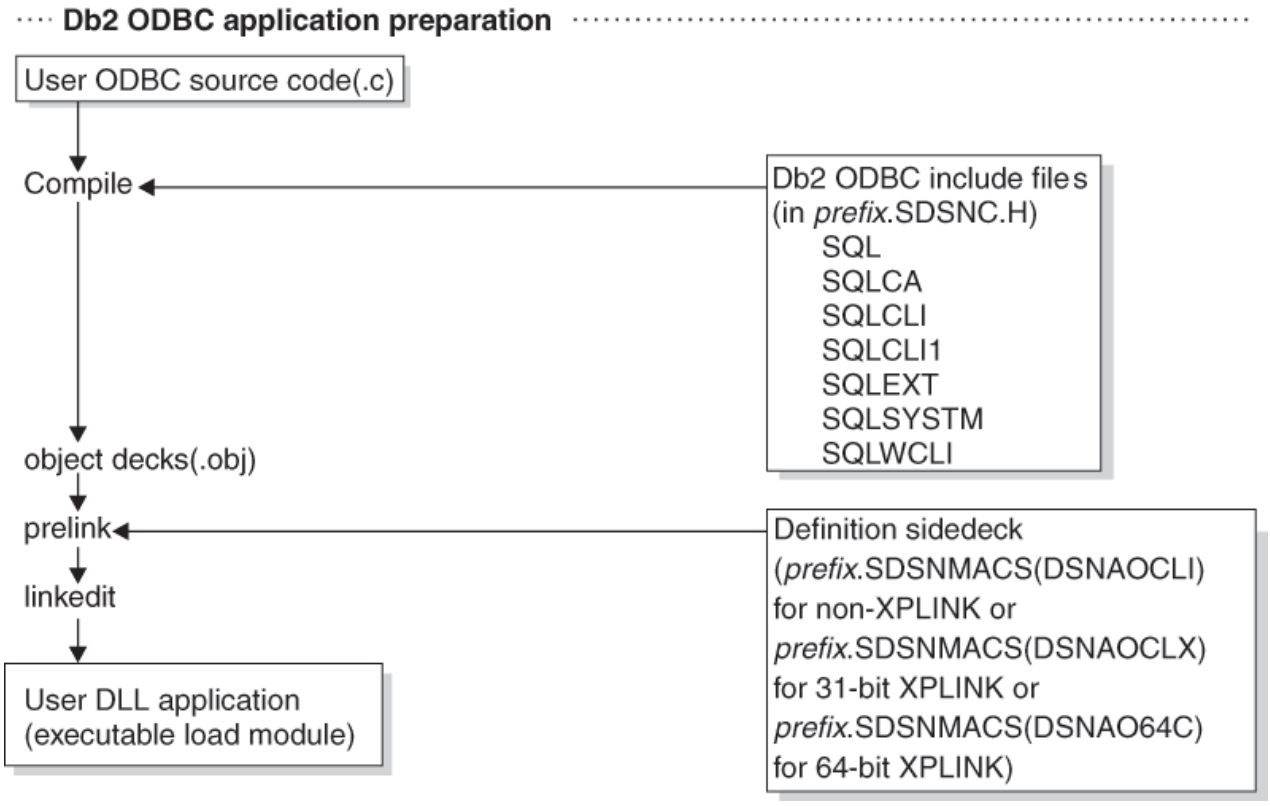



*Figure 8. Db2 ODBC program preparation*

**Related tasks**

Preparing and executing an ODBC application

You must compile, prelink, and link-edit an ODBC application before you can run it. Db2 ODBC provides sample programs to help you with program preparation.

**Related reference**

Db2 ODBC application requirements

To successfully build an ODBC application, you must ensure that the correct compile, prelink, and link-edit options are used. In particular, your application must generate the appropriate DLL linkage for the exported Db2 ODBC DLL functions.

## Db2 ODBC application requirements

To successfully build an ODBC application, you must ensure that the correct compile, prelink, and link-edit options are used. In particular, your application must generate the appropriate DLL linkage for the exported Db2 ODBC DLL functions.

Db2 ODBC applications have the following requirements:

- You must use a C or C++ compiler to compile the application. If you use a C compiler, you must specify the DLL compiler option.

  The C++ compiler always generates DLL linkage. However, the C compiler generates DLL linkage only 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.
- Language Environment base services must be installed on the subsystem or data sharing member.
- Applications must use the corresponding ODBC driver for the addressing mode in which they are running.

  Db2 ODBC applications can be written for either 31-bit addressing mode, AMODE(31) or 64-bit addressing mode, AMODE(64). 31-bit applications must use the 31-bit ODBC driver; they cannot use

the 64-bit ODBC driver. Likewise, 64-bit applications must use the 64-bit driver; they cannot use the 31-bit ODBC driver.

**Restriction:** Although 64-bit mode provides larger addressable storage, the amount of data that can be sent to or retrieved from Db2 by an ODBC application is still limited by the amount of storage that is available below the 2-GB bar. Therefore, for example, an application cannot declare a 2 GB LOB above the bar and insert the entire LOB value into a Db2 LOB column.

For ODBC applications that are built on z/OS UNIX, you do not need to copy the Db2 ODBC product file to HFS. You can directly reference the non-HFS Db2 ODBC data sets in the c89 compile command.

**Related concepts**

The Db2 ODBC run time environment

Db2 ODBC support is implemented as an IBM C/C++ Dynamic Load Library (DLL). All API calls are routed through the single ODBC driver that is loaded at run time into the application address space.

## Preparing and executing an ODBC application

You must compile, prelink, and link-edit an ODBC application before you can run it. Db2 ODBC provides sample programs to help you with program preparation.

### About this task

For guidance in how to prepare and execute an ODBC application, use the following online samples, which are provided in DSN1110.SDSNSAMP:

**DSN8O3VP**
A sample C application. You can use this sample to verify that your Db2 ODBC 3.0 installation is correct.

**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, prelink, link-edit, and execute the sample application DSN8O3VP or DSN8OIVP to use the non-XPLINK driver.

**DSNTEJ8X**
Sample JCL. You can use this sample to compile, link-edit, and execute the sample applications DSN8O3VP or DSN8OIVP to use the 31-bit XPLINK driver.

**DSNTEJ8E**
Sample JCL. You can use this sample to compile, link-edit, and execute the sample applications DSN8O3VP or DSN8OIVP to use the 64-bit XPLINK driver.

To use the DSN803VP or DSN8OIVP samples in z/OS UNIX, copy DSN803VP or DSN8OIVP from the sample data set to HFS. `user/db2` is considered the user's directory. For example:

```
oput  'DSN1110.SDSNSAMP(dsn8o3vp)' '/usr/db2/dsn8o3vp.c' TEXT
```

**Related concepts**

DSN8O3VP sample application

The DSN8O3VP sample program validates the installation of Db2 ODBC.

## Compiling an ODBC application

The first step in preparing an ODBC application is to compile it. the compiler process is slightly different depending on whether the application is an XPLINK application, a non-XPLINK application, or a 64-bit application.

### Before you begin

Before you begin compiling a Db2 ODBC application, ensure that you include the header file sqlcli1.h in your application. This header file contains all information that is required to compile the application. To include this file, add the following directive in the header of your Db2 ODBC application:

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

### Procedure

Depending on whether the application is an XPLINK application, a non-XPLINK application, or a 64-bit application, follow the appropriate instructions for compiling it.

For all Db2 ODBC applications, when you compile the application, you must add the DSN1110.SDSNC.H data set to the SYSPATH concatenation, the include path that is specified by the SEARCH or LSEARCH compiler options or the -I option for z/OS UNIX System Services. This data set includes all Db2 ODBC header files that define the function prototypes, constants, and data structures that are needed for a Db2 ODBC application.

#### Compiling non-XPLINK applications

If your application does not need to use z/OS XPLINK function linkage and is not written for 64-bit addressing, use the ODBC non-XPLINK driver and compile your application as a non-XPLINK application.

### Procedure

Perform one of the following actions:

- **To compile a non-XPLINK ODBC application in z/OS:** Specify the NOXPLINK compile option.

  For an example of a non-XPLINK compile job, see the DSNTEJ8 online sample in DSN1110.SDSNSAMP.
- **To compile a non-XPLINK ODBC C application in z/OS UNIX System Services:** Use the c89 compile command and specify the `-W 'c,dll'` compile option. (The `'dll'` option enables the use of the Db2 ODBC driver for C applications.)

  **Example:** To compile a C application that is 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"//'DSN1110.SDSNC.H'" \
   dsn8o3vp.c
  ```
- **To compile an ODBC C++ application in z/OS UNIX System Services:** Use the `cxx` compile command with the `-W 'c'` compile option

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

  ```
  cxx -c -W 'c,long,source,list' -
   -I"//'DSN1110.SDSNC.H'" \
   dsn8o3vp.c
  ```

### Compiling 31-bit XPLINK applications

If your 31-bit application needs to use z/OS XPLINK function linkage, use the ODBC 31-bit XPLINK driver and compile your application as an XPLINK application.

#### Procedure

Perform one of the following actions based on the compiler method:

| Compile method | Application language | Action |
|---|---|---|
| Compile on z/OS | All | Specify the XPLINK compile option.<br>**Example:** See DSNTEJ8X in *prefix*.SDSNSAMP. |
| Compile in z/OS UNIX System Services | C | Use the c89 command with the `-W 'c,xplink,dll'` compile option. (The `'dll'` option enables the use of the Db2 ODBC driver for C applications.)<br>**Example:** To compile a C application that is named dsn8o3vp.c that resides in the current working directory, use the following c89 compile command:<br>`c89 -c -W 'c,xplink,dll,long,source,list' -`<br>` -I"//'prefix.SDSNC.H'" \`<br>` dsn8o3vp.c` |
| | C++ | Use the cxx compile command with the `-W 'c,xplink'` compile option.<br>**Example:** To compile a C++ application that is named dsn8o3vp.c that resides in the current working directory, use the following cxx compile command:<br>`cxx -c -W 'c,xplink,long,source,list' -`<br>` -I"//'prefix.SDSNC.H'" \`<br>` dsn8o3vp.C` |
| Compile with the xlc utility on z/OS UNIX System Services | C | Specify the appropriate compiler options in the source program, a configuration file, or on the command line.<br>**Example:** To compile a C application that is named dsn8o3vp.c, you can use the following command:<br>`xlc -c -qxplink dsn8o3vp.c -I"//'prefix.SDSNC.H'"` |
| | C++ | Specify the appropriate compiler options in the source program, a configuration file, or on the command line.<br>**Example:** To compile a C++ application that is named dsn8o3vp.c, you can use the following command:<br>`xlC -c -qxplink dsn8o3vp.C -I"//'prefix.SDSNC.H'"` |

### Compiling 64-bit applications

If your application is written for 64-bit addressing, use the ODBC 64-bit driver to compile your application.

#### Procedure

Perform one of the following actions based on the compiler method:

| Compile method | Application language | Action |
|---|---|---|
| Compile on z/OS | All | Specify the LP64 compile option. Also consider specifying the FLOAT(HEX) and WARN64 compile options. [a] [b]<br><br>**Example:** See DSNTEJ8E in *prefix*.SDSNSAMP. |
| Compile in z/OS UNIX System Services | C | Use the c89 command with the `-W 'c,lp64'` compile option. Also consider specifying the `float(hex)` and `warn64` compile options. [a] [b]<br><br>**Example:** To compile a 64-bit C application that is named dsn8o3vp.c that resides in the current working directory, use the following c89 compile command:<br><br>`c89 -c -W 'c,lp64,float(hex),warn64,long,source,list'`<br>`-I"//'prefix.SDSNC.H'" \`<br>`dsn8o3vp.c` |
| | C++ | Use the `cxx` compile command with the `-W 'c,lp64'` compile option. Also consider specifying the `float(hex)` and `warn64` compile options. [a] [b]<br><br>**Example:** To compile a 64-bit C++ application that is named dsn8o3vp.c that resides in the current working directory, use the following `cxx` compile command:<br><br>`cxx -c -W 'c,lp64,float(hex),warn64,long,source,list'`<br>`-I"//'prefix.SDSNC.H'" \`<br>`dsn8o3vp.C` |
| Compile with the xlc utility on z/OS UNIX System Services | C | Specify the appropriate compiler options in the source program, a configuration file, or on the command line.<br><br>**Example:** The following example shows how to compile a 64-bit C application by using the command line options:<br><br>`xlc -c -q64 -qfloat=hex -qwarn64 dsn8o3vp.c`<br>`-I"//'prefix.SDSNC.H'"` |
| | C++ | Specify the appropriate compiler options in the source program, a configuration file, or on the command line.<br><br>**Example:** The following example shows how to compile a 64-bit C++ application by using the command line options:<br><br>`xlC -c -q64 -qfloat=hex -qwarn64 dsn8o3vp.C`<br>`-I"//'prefix.SDSNC.H'"` |

**Note:**

a. Specify FLOAT(HEX) if you want the 64-bit application to generate floating-point data in hexadecimal format. By default, specifying the LP64 compile option generates FLOAT(IEEE) code. This behavior is different from the default setting of FLOAT(HEX) when compiling 31-bit applications. For applications that are compiled with FLOAT(IEEE), you must convert floating-point data to hexadecimal format before it can be passed to ODBC for processing.

b. Specify WARN64 when recompiling existing 31-bit ODBC applications into 64-bit code. This option detects possible portability errors.

**Related reference**

Building and using Dynamic Link Libraries (DLLs) (XL C/C++ Programming Guide)

## Prelinking and link-editing an ODBC application

After you compile your ODBC application, you must prelink and link-edit it before you can run it. This process is slightly different depending on whether the application is an XPLINK application, a non-XPLINK application, or a 64-bit application.

### Procedure

Depending on whether the application is an XPLINK application, a non-XPLINK application, or a 64-bit application, follow the appropriate instructions for prelinking and link-editing.

You can prelink and link-edit XPLINK applications in one step. For non-XPLINK applications, you must use two steps.

Ensure that you include the appropriate Db2 ODBC definition sidedeck as input to the prelink or link-edit step of your application. Db2 for z/OS ODBC provides the following definition sidedecks:

- Non-XPLINK definition sidedeck, which defines all of the exported functions to use the non-XPLINK ODBC driver. This sidedeck resides in the DSN1110.SDSNMACS data set as member DSNOCLI.
- 31-bit XPLINK definition sidedeck, which defines all of the exported functions to use the 31-bit XPLINK ODBC driver. This sidedeck resides in the DSN1110.SDSNMACS data set as member DSNAOCLX.
- 64-bit XPLINK definition sidedeck, which defines all of the exported functions to use the 64-bit ODBC driver. This sidedeck resides in the DSN1110.SDSNMACS data set as member DSNAO64C.

The definition sidedeck that you include in the prelink or link-edit step of your application determines which Db2 ODBC dynamic load library is used.

#### Prelinking and link-editing non-XPLINK applications

Before you can link-edit a non-XPLINK application, you must prelink your application with a Db2 ODBC definition sidedeck.

### About this task

For non-XPLINK applications, you should use the non-XPLINK ODBC driver. Although doing so is not recommended, you can link-edit your non-XPLINK application as an XPLINK application.

### Procedure

Perform one of the following actions:

- **To link-edit a non-XPLINK ODBC application in z/OS:** Include the DSNAOCLI member as input to the prelinker by specifying it in the prelink SYSIN data definition statement concatenation.

  For an example of z/OS prelink and link-edit jobs, use the DSNTEJ8 and DSNTEJ8X sample jobs in DSN1110.SDSNSAMP.
- **To link-edit a non-XPLINK ODBC application in z/OS UNIX System Services:** Use the c89 command to prelink and link-edit C applications and the `cxx` command to prelink and link-edit C++ applications. Include the following items in the command:
  - The Db2 ODBC non-XPLINK definition sidedeck, DSN1110.SDSNMACS(DSNOCLI), as one of the input data sets.

    Before you can use a Db2 ODBC definition sidedeck 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 UNIX environment variable.
  - The `'dll'` link-edit option.

  **Example:** Assume that you have already compiled an application named myapp.c to create a myapp.o file in the current working directory and that you specified an alias that uses `.EXP` as the last qualifier

for the Db2 ODBC non-XPLINK definition sidedeck. You use the following c89 command to prelink and link-edit a C application:

```
c89  -W l,p,map,noer -W l,dll,AMODE=31,map \
 -o dsn8o3vp dsn8o3vp.o "//'DSN1110.SDSNC.EXP(DSNAOCLI)'"
```

You use the following cxx command to prelink and link-edit a C++ application:

```
cxx  -W l,p,map,noer -W l,dll,AMODE=31,map \
 -o dsn8o3vp dsn8o3vp.o "//'DSN1110.SDSNC.EXP(DSNAOCLI)'"
```

**Example:** Assume that you have already compiled an application named myapp.c to create a myapp.o file in the current working directory and that you changed the value of the _C89_XSUFFIX_HOST or _CXX_XSUFFIX_HOST environment variable to SDSNMACS. You use the following c89 command to prelink and link-edit a C application:

```
c89 -W l,p,map,noer -W l,dll,AMODE=31,map -o dsn8o3vp dsn8o3vp.o
"//'DSN1110.SDSNMACS(DSNAOCLI)'"
```

You use the following cxx command to prelink and link-edit a C++ application:

```
cxx -W l,p,map,noer -W l,dll,AMODE=31,map -o dsn8o3vp dsn8o3vp.o
"//'DSN1110.SDSNMACS(DSNAOCLI)'"
```

### Link-editing 31-bit XPLINK applications

For applications that are compiled as 31-bit XPLINK applications, you must link-edit your applications with the DFSMS binder.

#### About this task

For 31-bit XPLINK applications, you should use the 31-bit XPLINK ODBC driver. Although doing so is not recommended, you can also use the non-XPLINK driver with 31-bit XPLINK compiled applications.

#### Procedure

Perform one of the following actions:

- **To link-edit the application in z/OS:** Include the member DSNAOCLX as input to the binder by specifying it in the binder SYSIN data definition statement concatenation.

  For an example of z/OS XPLINK link-edit jobs, see the DSNTEJ8X sample job in DSN1110.SDSNSAMP.
- **To link-edit the application in z/OS UNIX System Services:** Use the c89 command to prelink and link-edit C applications and the cxx command to prelink and link-edit C++ applications. Include the following items in the command:
  - The Db2 ODBC definition sidedeck, DSN1110.SDSNMACS(DSNAOCLX), as one of the input data sets.

    Before you can use a Db2 ODBC definition sidedeck 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 UNIX environment variable.
  - The 'dll' link-edit option.

  Alternatively, you can use the xlc utility on z/OS UNIX System Services to link-edit the application. Follow the appropriate syntax for that utility.

  **Example of link-editing in z/OS UNIX when you specify an alias for the ODBC definition sidedeck:** Assume that you have already 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 sidedeck. You can use the following c89 command to prelink and link-edit an XPLINK 31-bit C application:

```
c89  -W l,xplink,dll,AMODE=31,map \
 -o dsn8o3vp dsn8o3vp.o "//'DSN1110.SDSNC.EXP(DSNAOCLX)'"
```

You can use the following cxx command to prelink and link-edit an XPLINK 31-bit C++ application:

```
cxx  -W l,xplink,dll,AMODE=31,map \
 -o dsn8o3vp dsn8o3vp.o "//'DSN1110.SDSNC.EXP(DSNAOCLX)'"
```

**Example of link-editing in z/OS UNIX when you change the value of the XSUFFIX_HOST variable:** Assume that you have already compiled an application named myapp.c to create a myapp.o file in the current working directory. Assume that you changed the value of the _C89_XSUFFIX_HOST or _CXX_XSUFFIX_HOST environment variable to SDSNMACS. You can use the following c89 command to prelink and link-edit an XPLINK 31-bit C application:

```
c89 -W l,xplink,dll,AMODE=31,map \
-o dsn8o3vp dsn8o3vp.o "//'DSN1110.SDSNMACS(DSNAOCLX)'"
```

You can use the following cxx command to prelink and link-edit an XPLINK 31-bit C++ application:

```
cxx -W l,xplink,dll,AMODE=31,map \
-o dsn8o3vp dsn8o3vp.o "//'DSN1110.SDSNMACS(DSNAOCLX)'"
```

**Related reference**

Building and using Dynamic Link Libraries (DLLs) (XL C/C++ Programming Guide)

**Link-editing 64-bit applications**

For 64-bit applications, you must link-edit your applications with the DFSMS binder. 64-bit applications are compiled as XPLINK applications.

**About this task**

For 64-bit applications, you must use the 64-bit ODBC driver.

**Procedure**

Perform one of the following actions:

- **To link-edit the application in z/OS:** Include the following items in your JCL:
  - The z/OS Language Environment library SCEEBND2 in the application's SYSLIB concatenation.
  - The 64-bit definition sidedeck, DSN1110.SDSNMACS(DSNAO64C), in the SYSLIN concatenation.
  - The definition sidedecks for the LE run time library bindings in the SYSLIN concatenation. For applications that are written in C, include the C RTL sidedeck CELQS003 from the LE SCEELIB data set. For applications that are written in C++, include both CELQS003 and the C++ sidedeck CELQSCPP.

  For an example of a z/OS link-edit job for a 64-bit application, see the DSNTEJ8E sample job in DSN1110.SDSNSAMP.
- **To link-edit the application in z/OS UNIX System Services:** Use the c89 command to prelink and link-edit C applications and the cxx command to prelink and link-edit C++ applications. Include the following items in the command:
  - The Db2 ODBC definition sidedeck for 64-bit applications, DSN1110.SDSNMACS(DSNAO64C), as one of the input data sets.

    Before you can use a Db2 ODBC definition sidedeck 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 UNIX environment variable.
  - The lp64 option

  Alternatively, you can use the xlc utility on z/OS UNIX System Services to link-edit the application. Follow the appropriate syntax for that utility.

  **Example of link-editing in z/OS UNIX when you specify an alias for the ODBC definition sidedeck:** Assume that you have already 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 sidedeck.

You use the following c89 command to prelink and link-edit a 64-bit C application:

```
c89 -W l,lp64,map -o dsn8o3vp dsn8o3vp.o "//'DSN1110.SDSNC.EXP(DSNAO64C)'"
```

You use the following `cxx` command to prelink and link-edit a 64-bit C++ application:

```
cxx -W l,lp64,map -o dsn8o3vp dsn8o3vp.o "//'DSN1110.SDSNC.EXP(DSNAO64C)'"
```

**Related reference**

Building and using Dynamic Link Libraries (DLLs) (XL C/C++ Programming Guide)

## Executing an ODBC application

You can execute an ODBC application by using a z/OS job or by using z/OS UNIX commands.

**Procedure**

Perform the following actions, as applicable:

- **For applications that you want to execute on z/OS:** Use one of the execution jobs in the following samples inDSN1110.SDSNSAMP as a model:

  **DSNTEJ8**
  Shows how to execute non-XPLINK applications

  **DSNTEJ8X**
  Shows how to execute 31-bit XPLINK applications.

  **DSNTEJ8E**
  Shows how to execute 64-bit XPLINK applications.

- **For applications that you want to execute on z/OS UNIX System Services:** Include the DSN1110.SDSNEXIT and DSN1110.SDSNLOAD data sets in the data set concatenation of your STEPLIB environment variable. You can set the STEPLIB environment variable in your `.profile` with the following statement:

  ```
  export STEPLIB=DSN1110.SDSNEXIT:DSN1110.SDSNLOAD
  ```

- **For all ODBC applications:** Ensure that your application can access the following items:
  - The DSN1110.SDSNLOAD data set

    The SDSNLOAD data set contains both the Db2 ODBC dynamic load library and the attachment facility module that is used to communicate with Db2.
  - Any site-specific DSNHDECP

    The Db2 for z/OS load module DSNHDECP contains, among other things, the coded character set ID (CCSID) information that Db2 for z/OS uses.

    A default DSNHDECP is shipped with Db2 for z/OS in the DSN1110.SDSNLOAD data set. However, if the values that are provided in the default DSNHDECP are not appropriate for your site, a new DSNHDECP can be created during the installation of Db2 for z/OS. If a site-specific DSNHDECP is created during installation, you should concatenate the data set that contains the new DSNHDECP before the DSN1110.SDSNLOAD data set in your STEPLIB or JOBLIB data definition statement.

- **For 31-bit XPLINK applications and 64-bit applications:** To use the 31-bit XPLINK driver or the 64-bit driver, you must perform the following actions:
  - Include z/OS Language Environment libraries SCEERUN and SCEERUN2 run time libraries in the application's STEPLIB, JOBLIB, LPALST, or LNKLST concatenation. For z/OS UNIX, ensure that these run time libraries are qualified with the prefix that the _C89_PLIB_PREFIX or _CXX_PLIB_PREFIX environment variable specifies.

- Ensure that the application can access the XPLINK dynamic link library DSNAOCLX in the DSN1110.SDSNLOD2 data set at execution time.
- For 31-bit applications, ensure that the application can access the XPLINK dynamic link library DSNAOCLX in the DSN1110.SDSNLOD2 data set at execution time.
- For 64-bit applications, ensure that the application can access the 64-bit dynamic link library DSNAO64C in the DSN1110.SDSNLOD2 data set at execution time.
- For non-XPLINK applications that use the XPLINK driver, specify the XPLINK(ON) Language Environment run time option to allocate XPLINK resources.

## How to define a subsystem to Db2 ODBC

You can define a Db2 subsystem in two ways. You can use MVSDEFAULTSSID, or you can use the default that is specified in the DSNHDECP load module.

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 DSN1110.SDSNEXIT data set. In this case, your STEPLIB DD card includes:

```
//STEPLIB    DD  DSN=DSN1110.SDSNEXIT,DISP=SHR
//           DD  DSN=DSN1110.SDSNLOAD,DISP=SHR
...
```

## Db2 ODBC initialization file

A set of optional keywords can be specified in a Db2 ODBC *initialization file*. An initialization file stores default values for various Db2 ODBC configuration options. Because the initialization file has EBCDIC text, you can use a file editor, such as the TSO editor, to edit it.

For most applications, use of the Db2 ODBC initialization file is not necessary. However, to make better use of Db2 for z/OS 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.

**Related concepts**

How to use the initialization file
The Db2 ODBC initialization file is read at application run time. You can specify the file by using either a DSNAOINI data definition statement or by defining a DSNAOINIz/OS UNIX environment variable.

**Related reference**

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

### How to use the initialization file

The Db2 ODBC initialization file is read at application run time. You can specify the file by using either a DSNAOINI data definition statement or by defining a DSNAOINIz/OS UNIX environment 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 environment 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=VB1A
/*
```

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 environment 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 environment 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 environment variable data set.

**Structure of the initialization file**

The initialization file consists of three types of section: common section, subsystem section, and data source sections.

The three sections, which are also called stanzas, are described as follows:

**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.

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.
- 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=VB1A
CONNECTTYPE=2
; Example SUBSYSTEM stanza for VB1A subsystem
[V81A]
MVSATTACHTYPE=CAF
PLANNAME=DSNACLI
; Example DATA SOURCE stanza for STLEC1 data source
[STLEC1]
AUTOCOMMIT=0
; Example DATA SOURCE stanza for STLEC1B data source
[STLEC1B]
CURSORHOLD=0
```

**Related reference**

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

## Db2 ODBC initialization keywords

Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

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

**ACCOUNTINGINTERVAL = COMMIT**
This keyword is placed in the subsystem section.

Use the ACCOUNTINGINTERVAL keyword to specify whether Db2 accounting records are produced at commit points. When ACCOUNTINGINTERNVAL is set to COMMIT, an accounting record is produced each time that a transaction is committed on a connection handle. When ACCOUNTINGINTERVAL is not specified, or a keyword value other than COMMIT is specified, accounting records are produced when the physical connection on the connection handle is terminated.

Db2 ODBC ignores ACCOUNTINGINTERVAL if MVSATTACHTYPE=CAF is specified, or if the Db2 ODBC application is running as a Db2 for z/OS stored procedure.

**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

**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.

**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 that the default of AUTOCOMMIT is on. Use extreme care when you override this default during run time. 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 that arise 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 situation 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 that are written by a third party, before you apply it to a DUW environment.

To enable global transaction processing in an application, specify AUTOCOMMIT=0, MULTICONTEXT=0, and MVSATTACHTYPE=RRSAF.

**BITDATA = 1 | 0**

This keyword is placed in the data source section.

You can use the BITDATA keyword to specify whether ODBC SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY, and SQL_BLOB types are reported as binary type data. IBM database servers

support columns with binary data types by defining CHAR, VARCHAR, and LONG VARCHAR columns with the FOR BIT DATA attribute, or by defining BINARY or VARBINARY columns.

Set BITDATA = 0 only if you are sure that all columns defined as FOR BIT DATA, BLOB, BINARY, or VARBINARY contain only character data, and the application is incapable of displaying binary data columns.

**1**
Report FOR BIT DATA, BLOB, BINARY, or VARBINARY data types as binary data. This value is the default.

**0**
Report FOR BIT DATA, BLOB, BINARY, or VARBINARY data types as character data.

**CLISCHEMA = *schema_name***

This keyword is deprecated.

If you specify this keyword, the ODBC driver does not use the database metadata stored procedures to retrieve catalog information.

**COLLECTIONID = *collection_id***
This keyword is placed in the data source section.

Specifies the collection identifier that is used to resolve the name of the package that is 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.

**CONCURRENTACCESSRESOLUTION = 0 | 1 | 2| 3**
The CONCURRENTACCESSRESOLUTION keyword controls how access to uncommitted data is resolved for a read transaction.

**0**
No setting. This value is the default value.

**1**
USE CURRENTLY COMMITTED. A read transaction can access the currently committed version of the data when the data is being updated or deleted. Rows that are in the process of being inserted are skipped. After this value is set, all user SELECT statements are prepared with the attribute USE CURRENTLY COMMITTED attribute. This option applies only when cursor stability (CS) or read stability (RS) isolation are in effect.

**2**
WAIT FOR OUTCOME. Read transactions that require access to data that is being updated or deleted must wait for a COMMIT or ROLLBACK operation to complete. Rows that are in the process of being inserted are not skipped. After this value is set, all user SELECT statements are prepared with the WAIT FOR COMMIT attribute.

**3**
SKIP LOCKED DATA. Read transactions can skip any rows that are incompatibly locked by other transactions. After this value is set, all user SELECT statements are prepared with the SKIP LOCKED attribute. This option applies only when cursor stability (CS) or read stability (RS) isolation are in effect.

When CONCURRENTACCESSRESOLUTION=1, the same rules and restrictions that apply to Db2 currently committed semantics also apply to ODBC. Access to currently committed data is supported for uncommitted INSERT and DELETE operation only in Version 10 and newer. Read transactions must still wait for uncommitted UPDATE operations to complete.

**CONNECTTYPE = 1 | 2**
This keyword is placed in the common section.

You can use the CONNECTTYPE keyword 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 value 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 | *ccsid***
This keyword is placed in the common section.

Use the CURRENTAPPENSCH keyword to specify either of the following items:

- The encoding scheme (Unicode, EBCDIC, or ASCII) that the ODBC driver uses for the following items:
  - Bound input or output host variables with the symbolic C data type SQL_C_CHAR
  - Character string arguments on a generic API call
- A CCSID value that provides the following information to the ODBC driver:
  - The same information that an encoding scheme provides. The ODBC driver derives the encoding scheme from the CCSID value.
  - The default values for the statement attributes SQL_CCSID_CHAR and SQL_CCSID_GRAPHIC. The CCSID value overrides the default CCSID settings in the DSNHDECP load module for application data with the SQL_C_CHAR data type.

The suffix-W APIs, which support UCS-2 string arguments, are not affected by CURRENTAPPENSCH. ODBC assumes UCS-2 for bound input or output host variables with the symbolic C data type SQL_C_WCHAR, regardless of the value of CURRENTAPPENSCH.

**Result of specifying an encoding scheme for CURRENTAPPENSCH:** When CURRENTAPPENSCH is set to EBCDIC, ASCII, or UNICODE, a SET CURRENT APPLICATION ENCODING SCHEME statement is sent to the data source after a successful connect. If this keyword is not present, the driver assumes EBCDIC as the default application encoding scheme.

**Result of specifying a CCSID for CURRENTAPPENSCH:** When CURRENTAPPENSCH is set to a CCSID value, the CCSID value provides default values to the statement attributes SQL_CCSID_CHAR and SQL_CCSID_GRAPHIC as follows:

- If *ccsid* is an SBCS CCSID, *ccsid* is the default value for SQL_CCSID_CHAR. SQL_CCSID_DEFAULT is the default value for SQL_CCSID_GRAPHIC.
- If *ccsid* is a DBCS CCSID, *ccsid* is the default value for SQL_CCSID_GRAPHIC. SQL_CCSID_DEFAULT is the default value for SQL_CCSID_CHAR.

  Specifying any of the UCS-2 CCSIDs (1200, 13488, or 17584) for *ccsid* is equivalent to specifying CURRENTAPPENSCH=UNICODE. If *ccsid* has any of those values, *ccsid* is not the default value for SQL_CCSID_GRAPHIC.
- If ccsid is a mixed CCSID, *ccsid* is the default value for SQL_CCSID_CHAR. The DBCS CCSID that is derived from ccsid is the default value for SQL_CCSID_GRAPHIC.

  If ccsid is 1208, 1208 is the default value for SQL_CCSID_CHAR. SQL_CCSID_DEFAULT is the default value for SQL_CCSID_GRAPHIC.

If an application calls `SQLSetStmtAttr()` to set the values for SQL_CCSID_CHAR or SQL_CCSID_GRAPHIC, those values override the default CCSID values that are set by CURRENTAPPENSCH.

**Result of not specifying CURRENTAPPENSCH or specifying an invalid value for CURRENTAPPENSCH:** When CURRENTAPPENSCH is not specified or the CURRENTAPPENSCH value is invalid, the ODBC driver uses 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. The SET CURRENT PATH SQL is used to set the value statement upon connection to the data source. Each schema name in the keyword string must be delimited with single quotation marks and separated by commas. The entire keyword string must be enclosed in double quotation marks 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 that each schema name begins the path, in the order that is shown in the default value. The order of the schema names in the path determines the order in which the names are resolved.

Unqualified user-defined functions, distinct types, and stored procedures are searched from the list of schemas that are 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 that is 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 database servers that support SET CURRENT SQLID (such as Db2 for z/OS). If this keyword is present, then a SET CURRENT SQLID statement is sent to the database server after a successful connect. Users and the application can 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:

- Application behavior does not depend on any information that is returned by `SQLGetInfo()` for SQL_CURSOR_COMMIT_BEHAVIOR or SQL_CURSOR_ROLLBACK_BEHAVIOR

- The application does not require cursors to be preserved from one transaction to the next.

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

**DB2EXPLAIN = 0 | 1 | 2 | 3**

This keyword is placed in the data source section.

The DB2EXPLAIN keyword sets the CURRENT EXPLAIN MODE special register to either YES or NO. You can specify one of the following values:

**0**

Sets the CURRENT EXPLAIN MODE special register to NO, which disables the EXPLAIN facility. In this case, Db2 does not capture any EXPLAIN information when explainable dynamic statements are executed. This value is the default.

**1**

Has the same effect as the value 0. This value is supported for Db2 family compatibility. For Db2 for Linux, UNIX, and Windows, the value 1 has a different meaning, but on Db2 for z/OS, the value 1 has the same meaning as the value 0.

**2**

Sets the CURRENT EXPLAIN MODE special register to YES, which enables the EXPLAIN facility. In this case, Db2 inserts EXPLAIN information into the EXPLAIN tables for explainable dynamic SQL statements.

**3**

Has the same effect as the value 2. This value is supported for Db2 family compatibility. For Db2 for Linux, UNIX, and Windows, the value 3 has a different meaning, but on Db2 for z/OS, the value 3 has the same meaning as the value 2.

Alternatively, you can set the CURRENT EXPLAIN MODE special register for ODBC applications by using the SQLSetConnectAttr() function with the SQL_ATTR_DB2EXPLAIN attribute or the SET CURRENT EXPLAIN MODE SQL statement.

If you want to set the CURRENT EXPLAIN MODE special register to EXPLAIN, you must use the SET CURRENT EXPLAIN MODE statement.

To get the EXPLAIN behavior that you want, you must also consider how you set the REOPT bind option.

**DBNAME = *dbname***

This keyword is placed in the data source section.

The DBNAME keyword is used only for connections to Db2 for z/OS, and only if (*base*) table catalog information is requested by the application.

If many tables exist in the Db2 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 that are returned to the application.

The value of the *dbname* keyword maps to the DBNAME column in the Db2 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 with SCHEMALIST and TABLETYPE to further limit the number of tables for which information is returned.

**DECIMALFLOATROUNDINGMODE=0 | 1 | 2 | 3 | 4 | 5 | 6**

This keyword is placed in the data source section.

DECIMALFLOATROUNDINGMODE specifies the rounding mode that is used when DECFLOAT data values are manipulated. If DECIMALFLOATROUNDINGMODE is present, Db2 ODBC sends a SET CURRENT DECFLOAT ROUNDING MODE statement to the data source after a successful connect. If DECIMALFLOATROUNDINGMODE is not present, ROUND_HALF_EVEN is assumed at the data source. Possible values and the corresponding CURRENT DECFLOAT ROUNDING MODE values are:

**0**
ROUND_HALF_EVEN

Round to the nearest integer. If the value is equidistant from two integers, round so that the final digit is even.

**1**
ROUND_HALF_UP

Round to the nearest integer. If the value is equidistant from two integers, round up.

**2**
ROUND_DOWN

Round toward 0. This is equivalent to truncation.

**3**
ROUND_CEILING

Round toward positive infinity.

**4**
ROUND_FLOOR

Round toward negative infinity.

**5**
ROUND_HALF_DOWN

Round to the nearest integer. If the value is equidistant from two integers, round down.

**6**
ROUND_UP

Round away from zero.

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

You can use the DIAGTRACE keyword to 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 by using the appropriate ODBC diagnostic trace 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 appropriate ODBC diagnostic trace command.

**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.

The *buffer size* value 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 (64 K). If the value specified is less than 65536, 65536 is used. The default value for the trace buffer size is 65536. For 64-bit applications, specify a *buffer size* value that is up to 10% bigger than what you would specify for 31-bit applications.

If a trace is active, this keyword is ignored.

**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.

**DIAGTRACE = 0 | 1**

This keyword is placed in the common section.

You can use the DIAGTRACE keyword to 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 by using the appropriate ODBC diagnostic trace 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 appropriate ODBC diagnostic trace command.

**DIAGTRACE_MASK = *.*.*.*.* | *trace_mask***

This keyword is placed in the common section.

The DIAGTRACE_MASK enables a trace mask, which limits the trace records collected by the Db2 ODBC diagnostic trace. Use this keyword only if a trace is started by the DIAGTRACE keyword.

The track mask consists of five parts that are delimited by periods.

- Types
- Products
- Components
- Functions
- Categories

Each part can consist of comma-separated lists, hyphen separated ranges, or single entries. The default setting of `DIAGTRACE_MASK=*.*.*.*.*` captures all trace records. To capture specific records, set the mask to the numbers that correspond to specific types, products, components, functions, and/or categories that you want to trace. If you want to trace all entry and exit records for component 41, set `DIAGTRACE_MASK=1,2.*.41.*.*` where 41 specifies the component, and 1 and 2 limit tracing to entry and exit records only.

The trace mask is intended for activating tracing for IBM debugging.

**EXTENDEDTABLEINFO = 0 | 1**

This keyword is placed in the data source section.

The EXTENDEDTABLEINFO keyword specifies whether information about extended table types is returned from a SQLTables() function call. Currently, there is one extended table type: ACCEL-ONLY TABLE.

Possible values are:

**0**

The result set that is returned by the SQLTables() function does not contain columns for extended table types.

Rows for extended table types are returned only if "TABLE" is explicitly specified in the *szTableType* parameter value in the SQLTables() call, or in the TABLETYPE initialization keyword, if *szTableType* is a null pointer. In this case, extended table types are listed as TABLE in the TABLE_TYPE column of the result set.

**1**

The result set that is returned by the SQLTables() function contains rows and columns for extended table types. In particular:

- The result set contains extra columns TEMPORAL_TABLE_TYPE, IS_ACCELERATED, ACCEL_ARCHIVE_STATUS, and IS_ARCHIVE_ENABLED after the columns that are always returned in the result set from SQLTables(). See "SQLTables() - Get table information" on page 405 for a description of those columns.
- Rows for extended table types are returned under the following circumstances:
  - All table types are implicitly requested by specifying a null pointer in the *szTableType* parameter of the SQLTables() call, and not specifying the TABLETYPE initialization keyword.
  - An extended table type name is explicitly specified in the *szTableType* parameter of the SQLTables() call, or in the TABLETYPE initialization keyword.

    In this case, the extended table type is listed by its extended table type name in the TABLE_TYPE column of the result set.

**GRANTEELIST = *userID1*, *userID2*, ... *userIDn***

This keyword is placed in the data source section.

You can use the GRANTEELIST keyword to reduce the amount of information that is returned when the application gets a list of privileges for tables, or privileges for columns in a table. The list of authorization IDs specified is used as a filter. If an application gets a list of privileges for a specific table, only the columns that have a privilege that is granted to the specified user IDs are returned.

This keyword is applicable to the APIs SQLColumnPrivileges and SQLTablePrivileges

**GRANTORLIST = *userID1*, *userID2*, ... *userIDn***

This keyword is placed in the data source section.

You can use the GRANTORLIST keyword to reduce the amount of information that is returned when the application gets a list of privileges for tables, or privileges for columns in a table. The list of authorization IDs specified is used as a filter. If the application gets a list of privileges for a specific table, only those columns that have a privilege that is granted by the specified user IDs are returned.

This keyword is applicable to the APIs SQLColumnPrivileges and SQLTablePrivileges

**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 that are 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.

**INTERRUPT = 0 | 1 | 2**

You can use the INTERRUPT keyword to specify the interrupt processing mode when SQLCancel() is called to cancel the processing on a statement.

**0**

Disable interrupt processing (SQLCancel() calls do not interrupt the processing.)

**1**

Interrupts are supported (default). In this mode, if interrupt is supported for the connection at the server, an interrupt is sent. Otherwise, the connection is dropped.

**2**

Interrupt drops the connection regardless of server's interrupt capabilities (`SQLCancel()` drops the connection.)

This keyword is applicable only to applications that have MVSATTACHTYPE=RRSAF specified in the initialization file. Only connections that are attached through the RRSAF attachment facility can be dropped.

When INTERRUPT is set to 1, Db2 ODBC always drops the connection that is associated with the statement.

**LIMITEDBLOCKFETCH = 0 | 1**

This keyword is placed in the data source section.

LIMITEDBLOCKFETCH specifies whether Db2 ODBC attempts to use limited block fetch when it fetches result sets from the connected data source. Limited block fetch can significantly reduce the number of trips to the Db2 server for data retrieval by grouping the rows that are retrieved by an SQL query into a block of rows in a query buffer. Limited block fetch benefits Db2 ODBC applications that retrieve large read-only result sets with forward-only cursors from a local Db2 server. LIMITEDBLOCKFETCH affects FETCH operations that are performed by the `SQLFetch()`, `SQLExtendedFetch()`, and `SQLFetchScroll()` functions. Possible values are:

**0**

Limited block fetch is not used.

**1**

Db2 ODBC attempts to use limited block fetch. If blocking is supported at the server for the result set that is being fetched, Db2 ODBC retrieves as many rows as it can fit in a query data block in a single fetch request. 1 is the default.

When you enable limited block fetch, you can also set the size of the query data block by setting the QUERYDATASIZE initialization parameter.

The specification of LIMITEDBLOCKFETCH=1 turns off any alternative fetch optimization that Db2 ODBC might otherwise use, such as Db2 multi-row fetch.

Db2 ODBC limited block fetch is not supported in the following cases:

- For connections to remote data sources
- For result sets other than read-only result sets
- For cursor types other than SQL_CURSOR_FORWARD_ONLY
- If any column in the result set is a file reference variable

- If the application sets the statement attribute SQL_NODESCRIBE to SQL_NODESCRIBE_ON and uses `SQLSetColAttributes()` to set the data source result descriptor for result set columns

If you enable limited block fetch for situations in which it is not supported, performance might be impacted.

**Related information:**

"ODBC limited block fetch" on page 434

**LITERALREPLACEMENT = 0 | 1**

This keyword is placed in the data source section.

The LITERALREPLACEMENT keyword specifies whether a dynamic SQL statement that contains literals is cached with the literal constants or with replacement markers for the literal constants. The default value is 0.

**0**

The statement is cached with the literal constants. If the statement contains one or more constants that are different from the cached version of the same dynamic statement, the statement is cached as a unique statement entry.

**1**

The statement is cached with replacement markers for literal constants. Db2 can share a cache entry for dynamic statements that are identical except for the literal constants if those statements also satisfy the following criteria:

- The statements do not contain parameter markers.
- The constants in the new statement can be reused in place of the constants in the cached statement.
- The statements satisfy all other conditions for dynamic statement cache sharing.

By sharing the dynamic cache entry, Db2 does not have to fully prepare the new statement, and the application performance might improve.

This keyword is equivalent to the CONCENTRATE STATEMENTS clause of the SQL PREPARE statement.

**MAXCONN = 0 | *positive number***

This keyword is placed in the common section.

The MAXCONN keyword is used to specify the maximum number of connections that are allowed for each Db2 ODBC application program. This can be used by an administrator as a governor for the maximum number of connections that are 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 | 2**

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 that are 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 use z/OS Context Services to create and manage its own contexts. 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.

To enable global transaction processing in an application, specify AUTOCOMMIT=0, MULTICONTEXT=0, and MVSATTACHTYPE=RRSAF.

**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 commit or roll back on a connection before it switches to another connection handle.

MULTICONTEXT=1 is not supported for applications that contain Db2 ODBC and embedded SQL.

**2**

MULTICONTEXT=2 and MULTICONTEXT=1 share connection characteristics. However, when MULTICONTEXT=2 is specified, Db2 ODBC enables a multithreaded application to always maintain an active environment handle under a designated Language Environment thread in a multiple-context environment. All restrictions that apply to MULTICONTEXT=1 also apply to MULTICONTEXT=2.

**Important:** Use MULTICONTEXT=2 only under the direction of IBM Software Support. Indiscriminate use of MULTICONTEXT=2 can cause applications to be abnormally terminated.

The application can use `SQLGetInfo()` with *InfoType* set to SQL_MULTIPLE_ACTIVE_TXN to determine whether MULTICONTEXT=1 or MULTICONTEXT=2 is supported.

MULTICONTEXT=1 and MULTICONTEXT=2 are ignored if any of these conditions are true:

- The application created a Db2 thread before it invoked Db2 ODBC. This situation is always the case for a stored procedure that uses Db2 ODBC.
- The application created and switched to a private context that uses z/OS Context Services before it invoked Db2 ODBC.
- The application started a unit of recovery with any RRS resource manager (for example, IMS) before it invoked Db2 ODBC.
- MVSATTACHTYPE=CAF is specified in the initialization file.
- The operating system level does not support Unauthorized Context Services.

**MVSATTACHTYPE = CAF | RRSAF**

This keyword is placed in the subsystem section.

The MVSATTACHTYPE keyword is used to specify the Db2 for z/OS attachment type that Db2 ODBC uses to connect to the Db2 for z/OS address space. This parameter is ignored if the Db2 ODBC application is running as a Db2 for z/OS ODBC stored procedure. In that case, Db2 ODBC uses the attachment type that was defined for the stored procedure.

**CAF**

Db2 ODBC uses the Db2 for z/OS call attachment facility (CAF). CAF is the default value.

**RRSAF**

Db2 ODBC uses the Db2 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 by using the z/OS Context Services. For more information, see “MULTICONTEXT = 0 | 1 | 2” on page 66.

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.

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

The MVSDEFAULTSSID keyword specifies the default Db2 subsystem that the application connects to when it invokes the SQLAllocHandle() function (with *HandleType* set to SQL_HANDLE_ENV). Specify the Db2 subsystem name, the subgroup attachment name, or group attachment name (if used in a data sharing group) to which connections are 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.

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

The PARAMOPTATOMIC keyword determines whether the underlying processing for multi-row inserts is done through atomic or non-atomic SQL. PARAMOPTATOMIC has the following values:

**0**
The underlying processing uses non-atomic SQL.

**1**
The underlying processing uses atomic SQL. This is the default.

**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**
SQLExecute() and SQLExecDirect() returns 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.

**301**
SQLError() and SQLGetDiagRec() returns more diagnostic information when a previous call to SQLExecute(), SQLExecDirect(), SQLFetch(), SQLExtendedFetch(), or SQLFetchScroll() on the same statement handle returned SQL_NO_DATA_FOUND (SQLCODE=100).

0: No workaround (default).
300

**PATCH2=300 behavior:** SQLExecute() and SQLExecDirect() 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.

The following table explains how PATCH2 settings affect return codes.

*Table 9. 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)<br>• 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)<br>• 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 = *plannname***

This keyword is placed in the subsystem section.

The PLANNAME keyword specifies the name of the Db2 for z/OS PLAN that was created during installation. A PLAN name is required when initializing the application connection to the Db2 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.

**QUERYDATASIZE = *integer***

This keyword is placed in the data source section.

QUERYDATASIZE specifies the amount of query data, in bytes, that Db2 returns on each FETCH operation when limited block fetch is enabled. The QUERYDATASIZE value can be used to optimize limited block fetch. It controls the number of trips to the data source that are required to retrieve data.

Using a larger value for QUERYDATASIZE can result in better performance. For example, if the result set size is 50 KB, and the value of QUERYDATASIZE is 32767 (32 KB), two trips to the data source are required to retrieve the result set. However, if QUERYDATASIZE is 65535 (62 KB), only one trip to the data source is required to retrieve the result set.

***integer***

One of the following QUERYDATASIZE values:

| | | | |
|---|---|---|---|
| 32767 | 294911 | 557055 | 819199 |
| 65535 | 327679 | 589823 | 851967 |
| 98303 | 360447 | 622591 | 884735 |
| 131071 | 393215 | 655359 | 917503 |
| 163839 | 425983 | 688127 | 950271 |
| 196607 | 458751 | 720895 | 983039 |
| 229375 | 491519 | 753663 | 1015807 |
| 262143 | 524287 | 786431 | 1048575 |

The default is 32767.

If you specify a value that is not a valid value, Db2 ODBC sets QUERYDATASIZE to the nearest valid value.

**RETURNALIASES = 0 | 1**

This keyword is placed in the data source section.

The RETURNALIASES keyword specifies whether aliases are included when you qualify rows for metadata procedures. If you exclude aliases and do not qualify them, you avoid costly joins with the base tables and improve performance.

**0**
Aliases are not considered when rows are qualified for metadata procedures

**1**
Aliases are considered when rows are qualified for metadata procedures

This keyword affects the following ODBC APIs.

- SQLColumns()
- SQLColumnPrivileges()
- SQLTables()
- SQLTablePrivileges()
- SQLStatistics()
- SQLSpecialColumns()
- SQLForeignKeys()
- SQLPrimaryKeys()

**RETCATALOGASCURRSERVER = 0 | 1**

This keyword is placed in the data source section.

You can use the RETCATALOGASCURRSERVER keyword to instruct the DBMS to return the CURRENT SERVER value instead of the null value for catalog columns.

**0**
Catalog functions return the null value for the catalog columns.

**1**
Catalog functions return the CURRENT SERVER value, instead of the null value, for the catalog columns.

This keyword affects the following ODBC APIs.

- SQLColumns()
- SQLColumnPrivileges()
- SQLTables()
- SQLTablePrivileges()
- SQLStatistics()
- SQLSpecialColumns()
- SQLForeignKeys()
- SQLPrimaryKeys()
- SQLProcedures()
- SQlProcedureColumns()

**RETURNSYSNONYMSCHEMA = 0 | 1**

This keyword is placed in the data source section.

The RETURNSYSNONYMSCHEMA controls whether the catalog APIs report the schema name for synonyms in the TABLE_SCHEM columns.

**0**
Catalog functions return NULL for the schema columns.

**1**
Catalog functions return the creator of the synonym for the schema columns.

This keyword affects the following ODBC APIs.

- SQLColumns()
- SQLColumnPrivileges()
- SQLTables()
- SQLTablePrivileges()
- SQLStatistics()
- SQLSpecialColumns()
- SQLForeignKeys()
- SQLPrimaryKeys()

**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 many 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 that are listed by the application. Each schema name is case-sensitive, must be delimited with single quotation marks and separated by commas. The entire string must also be enclosed in double quotation marks, for example:

```
SCHEMALIST="'USER1','USER2',USER3'"
```

For Db2 for z/OS ODBC, CURRENT SQLID can also be included in this list, but without the single quotation marks, for example:

```
SCHEMALIST="'USER1',CURRENT SQLID,'USER3'"
```

The maximum length of the keyword string is 2048 bytes.

This keyword can be used 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 database server. This improves performance of the table list retrieval in cases where the user is only interested in seeing the tables in a few schemas.

**STREAMBUFFERSIZE**

This keyword is placed in the data source section.

STREAMBUFFERSIZE is valid only when limited block fetch is enabled (LIMITEDBLOCKFETCH = 1). STREAMBUFFERSIZE specifies the threshold value, in bytes, that is used to determine whether LOB or XML data is returned in its entirety as inline data or as an internal token called a progressive reference:

- If the size of a LOB or XML object is less than or equal to the STREAMBUFFERSIZE value, the data value is returned inline as part of the row in the query block. The data value is cached on the ODBC driver for subsequent processing.
- If the size of a LOB or XML objects is greater than STREAMBUFFERSIZE, the data value is returned as a progressive reference.

The default value for STREAMBUFFERSIZE is 1,048,576 bytes, which is 1 MB.

**SYSSCHEMA = *sysschema***

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 following default values are used:

- SYSIBM on Db2 for z/OS
- SYSTEM on DB2 for VSE & VM
- QSYS2 on DB2 for i

This keyword can be used with SCHEMALIST, TABLETYPE (and DBNAME on Db2 for z/OS) to further limit the number of tables for which information is returned.

**TABLETYPE="'TABLE' | ,'ALIAS' | ,'VIEW' | , 'SYSTEM TABLE' | ,'SYNONYM' | 'GLOBAL TEMPORARY TABLE' |, 'AUXILIARY TABLE' |, 'MATERIALIZED QUERY TABLE' |, 'ACCEL-ONLY TABLE'"**

This keyword is placed in the data source section.

The TABLETYPE keyword specifies a list of one or more table types. If many 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 quotation marks, separated by commas, and in uppercase. The entire string must also be enclosed in double quotation marks, for example:

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

This keyword can be used 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 table names and associated information 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 supplies 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, the DBNAME keyword setting cannot be used to restrict information to a particular Db2 for z/OS subsystem.

'ACCEL-ONLY TABLE' is an extended table type name. Extended table type names are returned in the result set of an `SQLTables()` call only if the EXTENDEDTABLETYPE initialization parameter is set to 1.

**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. 1 is the default value.

**0**

The Db2 ODBC code is not threadsafe. This setting reduces the cost 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.

**TRACEPIDTID = 0 | 1**

This keyword is placed in the common section.

TRACEPIDTID is used only if a trace is started through the APPLTRACE keyword. When TRACEPIDTID is set to 1, the process ID and thread ID are added to the beginning of each line in the trace output. These IDs help you to differentiate the recorded information by process and thread when the Db2 ODBC application is running multiple concurrent Language Environment threads in a POSIX(ON) environment.

**TRACECTXTOKEN = 0 | 1**

This keyword is placed in the common section.

TRACECTXTOKEN is used only if a trace is started through the APPLTRACE keyword. When TRACECTXTOKEN is set to 1, the RRS context tokens are captured in the trace output. RSS context tokens help you to determine the execution path for applications that execute under different RRS contexts.

This keyword is applicable only to applications that create and manage their own RRS contexts with the z/OS Resource Recovery Services with keywords MVSATTACHTYPE=RRSAF and MULTICONTEXT=0 specified in the initialization file.

**TRACETIMESTAMP = 0 | 3**

This keyword is placed in the common section.

TRACETIMESTAMP is used only if a trace is started through the APPLTRACE keyword. When APPLTRACE is set to 1, the TRACETIMESTAMP keyword is used to capture different types of time stamp information in the Db2 ODBC application trace.

**0**

No time stamp information is written to the trace output.

**3**

An ISO time stamp is added to the beginning of each line in the trace output.

**TXNISOLATION = 1 | 2 | 4 | 8 | 32**

This keyword is placed in the data source section.

The TXNISOLATION keyword sets the isolation level to one of the following values:

**1**

Read uncommitted (uncommitted read)

**2**

Read committed (cursor stability) (default)

**4**

Repeatable read (read stability)

**8**

Serializable (repeatable read)

**32**
(No commit, DB2 for i only)

The words in round brackets are the Db2 equivalents for SQL92 isolation levels. "no commit" is not an SQL92 isolation level and is supported only on DB2 for i.

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

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

**1**
The underscore character (_) acts as a wildcard (default). The underscore is treated as a wildcard that matches 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 previous 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 previous example, `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.

**Related concepts**

Structure of the initialization file
The initialization file consists of three types of section: common section, subsystem section, and data source sections.

Multithreaded and multiple-context applications in Db2 ODBC
Db2 ODBC supports multi-threading and multiple contexts. You need to follow certain guidelines when using multiple contexts and multi-threading together in an application.

Global transactions in ODBC programs
A *global transaction* is a recoverable unit of work, or transaction, that is made up of changes to a collection of resources. You include global transactions in your application to access multiple recoverable resources in the context of a single transaction.

Impact of package bind options
When you bind ODBC packages, you must specify certain values for several bind options. You should also consider specific ODBC recommendations for several other bind options.

ODBC trace types
Db2 ODBC provides two traces that differ in purpose: an application trace for debugging user applications, and a service trace for problem diagnosis.

Conditions for statement sharing (Db2 Performance)

**Related tasks**

Accessing currently committed data to avoid lock contention (Db2 Performance)

Improving concurrency for applications that tolerate incomplete results (Db2 Performance)

**Related reference**

PREPARE (Db2 SQL)

optimize-clause (Db2 SQL)

## Database metadata stored procedures

When you install the ODBC drivers, you need to enable the metadata stored procedures.

The database metadata stored procedures must exist on every Db2 for z/OS data server to which your Db2 ODBC application connects. Db2 ODBC uses the database metadata stored procedures to retrieve catalog information and to implement the SQLGetTypeInfo() function. The database metadata stored procedures are installed as part of the Db2 for z/OS installation process.

**Related tasks**

Installation step 20: Set up Db2-supplied routines (Db2 Installation and Migration)

**Related reference**

Database metadata routines panel: DSNTIPRK (Db2 Installation and Migration)

## Migrating to the Db2 11 ODBC driver

When you migrate the ODBC driver for Db2 for z/OS from the Db2 10 driver to the Db2 11 driver, you must set up the Db2 11 ODBC run time environment.

### About this task

You must bind the Db2 11 ODBC DBRMs to each data source you want to migrate to the Db2 11 ODBC driver.

The online bind sample is available in DSN1110.SDSNSAMP(DSNTIJCL). You can use this bind sample as a guide for binding DBRMs to packages and binding an application plan.

### Procedure

To migrate to the Db2 11 ODBC driver:

1. Bind the DBRMs in DSNB10.SDSNDBRM to all data sources to which your ODBC applications connect. You must specify ENCODING(EBCDIC) when you bind the ODBC DBRMs to the local Db2 for z/OS subsystem.
2. Create at least one Db2 plan. Use the PKLIST keyword to specify all the packages that you create from the DBRMs

   Specify an appropriate ACTION parameter in the BIND PLAN statement:

   - If the plan does not exist, specify ACTION(ADD).

     If you specify ACTION(ADD), and the plan exists, the BIND command fails, and Db2 does not create a plan.
   - If the plan exists, and you want to retain the EXECUTE privilege on the plan for all users who already have that privilege, specify ACTION(REPLACE) RETAIN.
   - If the plan exists, and you want to revoke the EXECUTE privilege for everyone except the plan owner, specify ACTION(REPLACE). This is the default.

**Related concepts**

Impact of package bind options

When you bind ODBC packages, you must specify certain values for several bind options. You should also consider specific ODBC recommendations for several other bind options.

Db2 ODBC run time environment setup

The steps in setting up the Db2 ODBC run time environment must be performed once. These steps are performed as part of the installation process for Db2 for z/OS.

**Related tasks**

Binding DBRMs to create packages

You can bind database request modules (DBRMs) to create packages that use different isolation levels or default options.

Binding the application plan

When you bind the Db2 ODBC application plan, use the online bind sample, DSN1110.SDSNSAMP (DSNTIJCL), for guidance.

**Related information**

-130 (Db2 Codes)
-189 (Db2 Codes)
-805 (Db2 Codes)

# Migrating an ODBC 31-bit application to a 64-bit application

Consider migrating an application from 31-bit mode to 64-bit mode only if the application can take advantage of more than 2 GB of memory. Most applications run acceptably with the 31-bit addressing limitations.

### About this task

Applications that are most likely to benefit from 64-bit addressing are those that work with large amounts of data. For example, applications that work with large data sets can preload data into direct addressable memory for rapid access. Similarly, applications that work with large databases can cache more data in memory, which reduces the number of database requests.

Although 64-bit mode provides larger addressable storage, the amount of data that can be sent to and retrieved from Db2 by an ODBC application is still limited by the amount of storage that is available below the 2-GB bar. For example, an application cannot declare a 2-GB LOB above the bar and insert the entire LOB value into a Db2 LOB column.

### Procedure

To migrate an ODBC 31-bit application to a 64-bit application:

1. Modify your application as follows:
   - If your application calls the SQLSetConnectOption function, change the application to use the SQLSetConnectAttr function instead. SQLSetConnectOption is not supported in the ODBC 64-bit driver.
   - If your application calls the SQLSetStmtOption function, change the application to use the SQLSetStmtAttr function instead. SQLSetStmtOption is not supported in the ODBC 64-bit driver.
   - Use the C-defined types SQLINTEGER and SQLUINTEGER to declare all Db2 ODBC variables and arguments that contain 32-bit integer values.

   **Recommendation:** If your application calls any ODBC functions that have arguments of type SQLINTEGER or SQLUINTEGER, change those function calls to pass and receive values of type SQLLEN instead of SQLINTEGER and type SQLULEN instead of SQLUINTEGER.
2. Recompile the application with the LP64 compile option. If you want the compiler to identify any potential portability errors, also specify the WARN64 compile option.
3. Link-edit the application with the definition sidedeck for the z/OS ODBC 64-bit driver, DSN1110.SDSNMACS(DSNAO64C).
4. Execute the application.

**Related tasks**

Preparing and executing an ODBC application
You must compile, prelink, and link-edit an ODBC application before you can run it. Db2 ODBC provides sample programs to help you with program preparation.

## Example 64-bit ODBC application

Applications that deal with large amounts of data are good candidates for 64-bit addressing.

The following example shows a 64-bit application that inserts a row into a table by binding two application variables to INTEGER and CHAR(10) parameter markers and then uses SQLFetch() to retrieve

the row data from bound columns of the result set. The DSNTEJ8E sample in the SDSNSAMP library shows the JCL for compiling, binding, and executing a 64-bit ODBC application.

```
/* Declare local variables. When compiled in LP64 mode, variables are
   allocated in stack storage above the 2G line */
SQLINTEGER         H1INT;
SQLCHAR             H1CHAR[10];
SQLINTEGER         H2INT;
SQLCHAR             H2CHAR[10];

SQLLEN             LEN_H1INT;
SQLLEN             LEN_H1CHAR;
SQLLEN             LEN_H2INT;
SQLLEN             LEN_H2CHAR;

strcpy( (char *)sqlstmt, "INSERT INTO MYTABLE (INT4, CHAR10) VALUES( ?, ? )" );
rc = SQLPrepare( hstmt, sqlstmt, SQL_NTS );

/* Bind to DB2 INTEGER with data located above the 2G line*/
rc = SQLBindParameter( (SQLHSTMT) hstmt,
                       (SQLUSMALLINT) 1,
                       (SQLSMALLINT) SQL_PARAM_INPUT,
                       (SQLSMALLINT) SQL_C_LONG,
                       (SQLSMALLINT) SQL_INTEGER,
                       (SQLULEN) 0,
                       (SQLSMALLINT) 0,
                       (SQLPOINTER) &H1INT,
                       (SQLLEN) sizeof(H1INT),
                       (SQLLEN *) &LEN_H1INT );

/* Bind to DB2 CHAR(10) with data located above the 2G line*/
rc = SQLBindParameter( (SQLHSTMT) hstmt,
                       (SQLUSMALLINT) 2,
                       (SQLSMALLINT) SQL_PARAM_INPUT,
                       (SQLSMALLINT) SQL_C_CHAR,
                       (SQLSMALLINT) SQL_CHAR,
                       (SQLULEN) 10,
                       (SQLSMALLINT) 0,
                       (SQLPOINTER) H1CHAR,
                       (SQLLEN) sizeof(H1CHAR),
                       (SQLLEN *) &LEN_H1CHAR );

rc = SQLExecute( hstmt );
.
.
.
strcpy( (char *)sqlstmt, "SELECT INT4, CHAR10 FROM MYTABLE" );

/* Bind DB2 INTEGER column */
rc = SQLBindCol( (SQLHSTMT) hstmt,
                 (SQLUSMALLINT) 1,
                 (SQLSMALLINT) SQL_C_LONG,
                 (SQLPOINTER) &H2INT,
                 (SQLLEN) sizeof(H2INT),
                 (SQLLEN *) &LEN_H2INT );

/* Bind DB2 CHAR(10) column */
rc = SQLBindCol( (SQLHSTMT) hstmt,
                 (SQLUSMALLINT) 2,
                 (SQLSMALLINT) SQL_C_CHAR,
                 (SQLPOINTER) H2CHAR,
                 (SQLLEN) sizeof(H2CHAR),
                 (SQLLEN *) &LEN_H2CHAR );

/* Fetch data into storage above the 2G line */
rc = SQLFetch( hstmt );
.
.
.
```

# Chapter 4. ODBC functions

Db2 ODBC provides various SQL-related functions with unique purposes, diagnostics, and restrictions.

**About these topics**

These topics might contain any of the following sections, in addition to other sections. Certain sections are omitted for deprecated functions.

**Purpose**

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. The following table 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 10. Sample function specification table*

| ODBC | X/OPEN CLI | ISO CLI |
|---|---|---|
| 3.0 | Yes | Yes |

**Syntax**

Contains a generic C language prototype for the function.

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**

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

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.

**Return codes**

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()`

**Diagnostics**

Contains SQLSTATEs that are explicitly returned by Db2 ODBC and indicates the cause of the error. (Db2 ODBC can also return SQLSTATEs that the database management system generates.) To obtain these SQLSTATE values, call `SQLGetDiagRec()` on a function that returns SQL_ERROR or SQL_SUCCESS_WITH_INFO.

**Related concepts**

Diagnostics

Diagnostics deal with warning or error conditions that are generated within an application.

The Db2 ODBC run time environment

© Copyright IBM Corp. 1997, 2020

Db2 ODBC support is implemented as an IBM C/C++ Dynamic Load Library (DLL). All API calls are routed through the single ODBC driver that is loaded at run time into the application address space.

**Related reference**

Deprecated ODBC functions
Db2 ODBC supports the ODBC 3.0 standard and all deprecated functions. Use the function replacements where applicable to optimize performance.

**Related information**

Microsoft open database connectivity (ODBC)

## Status of support for ODBC functions

Each function has its own ODBC 3.0 conformance level, and Db2 ODBC support level, and certain functions are deprecated.

Each of the following tables provides a list of functions that support a particular task. The tables indicate the level of Db2 ODBC support and and Microsoft ODBC 3.0 conformance level for each function.

The table contains the following values, by column:

**ODBC 3.0 level**
The values of this column have the following meanings:

**No**
Indicates that the function is not supported by ODBC 3.0.

**Deprecated**
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.

**Level 1**
Indicates that the function is part of the ODBC 3.0 Level 1 conformance level.

**Level 2**
Indicates that the function is part of the ODBC 3.0 Level 2 conformance level.

**Db2 ODBC support**
The values of this column have the following meanings:

**No**
Indicates that the function is not supported by Db2 ODBC.

**Deprecated**
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.

**Connecting to a data source**

*Table 11. Functions for connecting to a data source*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| `SQLAllocConnect()` | Deprecated | Deprecated | Obtains a connection handle. |
| `SQLAllocEnv()` | Deprecated | Deprecated | Obtains an environment handle. One environment handle is used for one or more connections. |
| `SQLAllocHandle()` | Core | Current | Obtains a handle. |

*Table 11. Functions for connecting to a data source (continued)*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLBrowseConnect() | Level 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 to a specific driver by data source name, user ID, and password. |
| SQLDriverConnect() | Core | Current | Connects to a specific driver with a connection string.<br><br>**IBM specific:** This function is also extended with the additional IBM keywords that are supported in the ODBC.INI file in the Db2 for Linux, UNIX, and Windows CLI environment. Within the Db2 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**

*Table 12. Functions for obtaining information about a driver and data source*

| Function name | ODBC 3.0 level | Status in Db2 ODBC | Purpose |
|---|---|---|---|
| SQLDataSources() | Core | Current | Returns the list of available data sources. |
| 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 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**

*Table 13. Functions for setting and retrieving driver attributes*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLGetConnectAttr() | Core | Current | Returns the value of a connection attribute. |
| SQLGetConnectOption() | Deprecated | Deprecated | 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. |

*Table 13. Functions for setting and retrieving driver attributes (continued)*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLGetStmtOption() | Deprecated | Deprecated | Returns the value of a statement attribute. |
| SQLSetConnectAttr() | Core | Current | Sets a connection attribute. |
| SQLSetConnectOption() | Deprecated | Deprecated | Sets a connection attribute. |
| SQLSetEnvAttr() | Core | Current | Sets an environment attribute. |
| SQLSetStmtAttr() | Core | Current | Sets a statement attribute. |
| SQLSetStmtOption() | Deprecated | Deprecated | Sets a statement attribute. |

**Setting and retrieving descriptor fields**

*Table 14. Functions for setting and retrieving descriptor fields*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| 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**

*Table 15. Functions for preparing SQL requests*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLAllocStmt() | Deprecated | Deprecated | Allocates a statement handle. |
| SQLBindFileToParam() | No | Current | Associates a parameter marker in an SQL statement with a file reference or an array of file references. |
| 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() | Deprecated | 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. |

*Table 15. Functions for preparing SQL requests (continued)*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLSetParam() | Deprecated | Deprecated | Assigns storage for a parameter in an SQL statement (ODBC 2.0). In ODBC 3.0, SQLBindParameter() replaces this function. |
| SQLSetScrollOptions() | Deprecated | No | Sets attributes that control cursor behavior. In ODBC 3.0, SQLGetInfo() and SQLSetStmtAttr() replace this function. |

**Submitting requests**

*Table 16. Functions for submitting requests*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| 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 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**

*Table 17. Functions for retrieving results and information about results*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLBindCol() | Core | Current | Assigns storage for a result column and specifies the data type. |
| SQLBindFileToCol() | No | Current | Associates a LOB column in a result set to a file reference or an array of file references. |
| SQLBulkOperations() | Level 1 | Current | Performs bulk inserts operations. |
| SQLColAttribute() | Core | Current | Describes attributes of a column in the result set. |
| SQLColAttributes() | Deprecated | Deprecated | Describes attributes of a column in the result set. |
| SQLDescribeCol() | Core | Current | Describes a column in the result set. |

*Table 17. Functions for retrieving results and information about results (continued)*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLError() | Deprecated | Deprecated | Returns additional error or status information. |
| SQLExtendedFetch() | Deprecated | Current | Returns multiple result rows. |
| SQLFetch() | Core | Current | Returns a result row. |
| SQLFetchScroll() | Core | Current | Returns row sets that are specified by absolute or relative position. |
| 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() | Level 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, delete, or merge request. |
| SQLSetColAttributes() | No | Current | Sets attributes of a column in the result set. |
| SQLSetPos() | Level 1 | Current | Allows an application to refresh, update, delete, and insert rows in the rowset |

**Handling large objects**

*Table 18. Functions for handling large objects*

| Function name | ODBC 3.0 level | Db2 ODBC support | 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**

*Table 19. Functions for obtaining catalog information*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLColumnPrivileges() | Level 2 | Current | Returns a list of columns and associated privileges for a table. |

*Table 19. Functions for obtaining catalog information (continued)*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLColumns() | Core | Current | Returns the list of column names in specified tables. |
| SQLForeignKeys() | Level 2 | Current | Returns a list of column names that comprise foreign keys, if they exist, for a specified table. |
| SQLPrimaryKeys() | Level 1 | Current | Returns the list of column names that comprise the primary key for a table. |
| SQLProcedureColumns() | Level 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() | Level 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() | Level 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**

*Table 20. Functions for terminating a statement*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| 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() | Deprecated | Deprecated | Commits or rolls back a transaction. |

**Terminating a connection**

*Table 21. Functions for terminating a connection*

| Function name | ODBC 3.0 level | Db2 ODBC support | Purpose |
|---|---|---|---|
| SQLDisconnect() | Core | Current | Closes the connection. |
| SQLFreeConnect() | Deprecated | Deprecated | Releases the connection handle. |
| SQLFreeEnv() | Deprecated | Deprecated | Releases the environment handle. |
| SQLFreeHandle() | Core | Current | Releases an environment, connection, statement, or descriptor handle. |

**ODBC 3.0 functions that are not supported by Db2 ODBC**

The following ODBC 3.0 functions are not supported by Db2 ODBC:

- SQLBrowseConnect().
- 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.
- 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.
- SQLSetScrollOptions(). This function is superseded by the following statement attributes:
  - SQL_ATTR_CURSOR_TYPE
  - SQL_ATTR_CONCURRENCY
  - SQL_KEYSET_SIZE
  - SQL_ATTR_ROWSET_SIZE

**Related reference**

Deprecated ODBC functions

Db2 ODBC supports the ODBC 3.0 standard and all deprecated functions. Use the function replacements where applicable to optimize performance.

# SQLAllocConnect() - Allocate a connection handle

SQLAllocConnect() is a deprecated function and is replaced by SQLAllocHandle().

**ODBC specifications for SQLAllocConnect()**

*Table 22. SQLAllocConnect() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | Yes |

**Syntax**

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

**Function arguments**

The following table lists the data type, use, and description for each argument in this function.

*Table 23. SQLAllocConnect() arguments*

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

**Related reference**

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

## SQLAllocEnv() - Allocate an environment handle

`SQLAllocEnv()` is a deprecated function and is replaced by `SQLAllocHandle()`.

**ODBC specifications for SQLAllocEnv()**

*Table 24. SQLAllocEnv() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | Yes |

**Syntax**

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

**Function arguments**

The following table lists the data type, use, and description for each argument in this function.

*Table 25. SQLAllocEnv() arguments*

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

**Related reference**

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

## SQLAllocHandle() - Allocate a handle

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

**ODBC specifications for SQLAllocHandle()**

*Table 26. SQLAllocHandle() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

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

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 27. 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:<br>• SQL_HANDLE_ENV for an environment handle<br>• SQL_HANDLE_DBC for a connection handle<br>• 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:<br>• SQL_NULL_HANDLE (or ignore this argument) if you are allocating an environment handle<br>• To the environment handle if you are allocating a connection handle<br>• 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.

  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.

  The Db2 ODBC 3.0 driver does not support multiple environments.

- **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.

  **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

### 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().

The following table lists each SQLSTATE that this function generates with a description and explanation for each value.

*Table 28. SQLAllocHandle() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (SQLAllocHandle() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **08**003 | 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 for z/OS subsystem or any other system-related error. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is unable to allocate memory for the specified handle. |
| **HY**009 | Invalid use of a null pointer. | The *OutputHandlePtr* argument specifies a null pointer |

*Table 28. SQLAllocHandle() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**013 | 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. |
| **HY**014 | 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. |
| **HY**092 | Option type out of range. | The *HandleType* argument does not specify one of the following values:<br>• SQL_HANDLE_ENV<br>• SQL_HANDLE_DBC<br>• 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 the DSN803VP sample application or DSN8O3VP in the DSN1110.SDSNSAMP data set.

**Related concepts**

ODBC 3.0 driver behavior
Behavioral changes refer to functionality that varies depending on the version of ODBC that is in use. The ODBC 2.0 and ODBC 3.0 drivers behave according to the setting of the SQL_ATTR_ODBC_VERSION environment attribute.

Db2 ODBC initialization file
A set of optional keywords can be specified in a Db2 ODBC *initialization file*. An initialization file stores default values for various Db2 ODBC configuration options. Because the initialization file has EBCDIC text, you can use a file editor, such as the TSO editor, to edit it.

DSN8O3VP sample application
The DSN8O3VP sample program validates the installation of Db2 ODBC.

Multithreaded and multiple-context applications in Db2 ODBC
Db2 ODBC supports multi-threading and multiple contexts. You need to follow certain guidelines when using multiple contexts and multi-threading together in an application.

Problem diagnosis
Several guidelines exist for working with the Db2 ODBC traces, including information about general diagnosis, debugging, and abnormal terminations.

**Related reference**

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLFreeHandle() - Free a handle
`SQLFreeHandle()` frees an environment handle, a connection handle, or a statement handle.

SQLGetDiagRec() - Get multiple field settings of diagnostic record
`SQLGetDiagRec()` returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information. `SQLGetDiagRec()` also returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the error message text.

# SQLAllocStmt() - Allocate a statement handle

`SQLAllocStmt()` is a deprecated function and is replaced by `SQLAllocHandle()`.

### ODBC specifications for SQLAllocStmt()

*Table 29. SQLAllocStmt() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | Yes |

### Syntax

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

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 30. SQLAllocStmt() arguments*

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

### Related reference

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

# SQLBindCol() - Bind a column to an application variable

`SQLBindCol()` binds a column to an application variable. You can call `SQLBindCol()` once for each column in a result set from which you want to retrieve data or LOB locators.

### ODBC specifications for SQLBindCol()

*Table 31. SQLBindCol() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

For 31-bit applications, use the following syntax:

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

For 64-bit applications, use the following syntax:

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

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 32. 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 32. 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:<br>• SQL_C_BIGINT<br>• SQL_C_BINARY<br>• SQL_C_BINARYXML<br>• SQL_C_BIT<br>• SQL_C_BLOB_LOCATOR<br>• SQL_C_CHAR<br>• SQL_C_CLOB_LOCATOR<br>• SQL_C_DBCHAR<br>• SQL_C_DBCLOB_LOCATOR<br>• SQL_C_DECIMAL64<br>• SQL_C_DECIMAL128<br>• SQL_C_DOUBLE<br>• SQL_C_FLOAT<br>• SQL_C_LONG<br>• SQL_C_SHORT<br>• SQL_C_TYPE_DATE<br>• SQL_C_TYPE_TIME<br>• SQL_C_TYPE_TIMESTAMP<br>• SQL_C_TYPE_TIMESTAMP_EXT<br>• SQL_C_TINYINT<br>• SQL_C_WCHAR<br>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. |
| 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.<br>If the *rgbValue* argument is null, the column is unbound. |
| SQLINTEGER (31-bit) or SQLLEN (64-bit) [1] | *cbValueMax* | input | Specifies the size of the *rgbValue* buffer in bytes that are available to store the column data or the LOB locator.<br>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. |

*Table 32. SQLBindCol() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLINTEGER * (31-bit) or SQLLEN * (64-bit) [1] | *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.<br><br>SQLFetch() returns SQL_NULL_DATA in this argument if the data value of the column is null.<br><br>This pointer value can be null. If this pointer is not null, it must be unique for each bound column.<br><br>SQL_NO_LENGTH can also be returned. See "Usage" on page 95 for more details. |

**Notes:**

1. For 64-bit applications, the data type SQLINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLLEN is recommended.

**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" on page 95 for more details.

### Usage

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 database management system 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.
- A LOB locator, for LOB columns. When you bind to LOB locators, the locator, not the data itself, is transferred from the database management system 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 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 might need to obtain column attributes before you call SQLBindCol(). To obtain these attributes, call SQLDescribeCol() or SQLColAttribute().

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. 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. 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.

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().

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.

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 database management system 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, the CCSID that is specified for the CURRENTAPPENSCH keyword is an EBCDIC CCSID, 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 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, or the CCSID that is specified for the CURRENTAPPENSCH keyword is a Unicode CCSID (1200, 1208, 13488 or 17584).
  - 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, or the CCSID that is specified for the CURRENTAPPENSCH keyword is an ASCII CCSID.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLBindCol() call.

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 server 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

#### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 33. SQLBindCol() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | 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. | The specified value for the *icol* argument is less than 0 or greater than the number of columns in the result set. |
| **HY**003 | Program type out of range. | The *fCType* argument is not a valid data type or SQL_C_DEFAULT. |
| **HY**010 | 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.) |
| **HY**013 | 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. |
| **HY**090 | Invalid string or buffer length. | The specified value for the *cbValueMax* argument is less than 0. |
| **HY**C00 | 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.

#### Example

Refer to SQLExtendedFetch() for more details. This function returns a block of data containing multiple rows for each bound column.

**Related concepts**

ODBC programming hints and tips
When you program a Db2 ODBC application, you can avoid common problems, improve performance, reduce network flow, and maximize portability.

Data types and data conversion
When you write a Db2 ODBC application, you must work with both SQL data types and C data types. The database server uses SQL data types, and the application uses C data types.

Application encoding schemes and Db2 ODBC
Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The Db2 ODBC driver supports input and output character string arguments to ODBC APIs and input and output host variable data in each of these encoding schemes.

Retrieval of a result set into an array
An application can issue a query statement and fetch rows from the result set that the query generates.

Code ODBC functions for efficient data retrieval
You can retrieve data more efficiently by following several guidelines.

Example of binding result set columns to retrieve UCS-2 data
You can use SQLBindCol() to bind the first column of a result set to a Unicode UCS-2 application buffer.

**Related reference**

C and SQL data types
Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

SQLExtendedFetch() - Fetch an array of rows
SQLExtendedFetch() extends the function of SQLFetch() by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLBindFileToCol() - Associate a column with a file reference

SQLBindFileToCol() associates a LOB column in a result set to a file reference or an array of file references. This association enables data in that column to be transferred directly into a file when each row is fetched for the statement handle.

### ODBC specifications for SQLBindFileToCol()

*Table 34. SQLBindFileToCol() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| No | No | No |

### Syntax

```
SQLRETURN   SQLBindFileToCol (SQLHSTMT          StatementHandle,   /* hstmt */
                              SQLUSMALLINT      ColumnNumber,      /* icol */
                              SQLCHAR           *FileName,
                              SQLSMALLINT       *FileNameLength,
                              SQLUINTEGER       *FileOptions,
                              SQLSMALLINT       MaxFileNameLength,
```

```
                                SQLINTEGER        *StringLength,
                                SQLINTEGER        *IndicatorValue);
```

**Function arguments**

The data type, use, and description for each argument in this function are similar to those of `SQLBindCol()` arguments.

*Table 35. SQLBindFileToCol arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *StatementHandle* | input | Statement handle. |
| SQLUSMALLINT | *icol* | input | Number that identifies the column. Columns are numbered sequentially, from left to right, starting at 1. |
| SQLCHAR * | *FileName* | input (deferred) | Pointer to the location that will contain the file name or an array of file names at the time of the next fetch using *StatementHandle*. The name is the absolute path name of each file.<br>This pointer cannot be NULL. |
| SQLSMALLINT * | *FileNameLength* | input (deferred) | Pointer to the location that will contain the length of the file name or an array of lengths at the time of the next fetch using *StatementHandle*. If this pointer is NULL, ODBC treats *FileName* as a null-terminated string. The result is the same as if a length of SQL_NTS is passed.<br>The maximum value of the file name length is 255. |
| SQLUINTEGER * | *FileOptions* | input (deferred) | Pointer to the location that will contain the file option or an array of file options to be used when data is written to the file at the time of the next fetch using *StatementHandle*. The following *FileOptions* values are supported:<br>**SQL_FILE_CREATE**<br>Create a new file. If a file with this name already exists, SQL_ERROR is returned.<br>**SQL_FILE_OVERWRITE**<br>If the file already exists, overwrite it. Otherwise, create a new file.<br>**SQL_FILE_APPEND**<br>If the file already exists, append the data to it. Otherwise, create a new file.<br>Only one option can be specified for a file. There is no default. |
| SQLSMALLINT | *MaxFileNameLength* | input | Specifies the length of the *FileName* buffer. If the application uses `SQLExtendedFetch()` to retrieve multiple rows for the LOB column, specifies the length of each element in the *FileName* array. |
| SQLINTEGER * | *StringLength* | output (deferred) | Pointer to the location that contains the length or array of lengths of the LOB data that is returned. If this pointer is NULL, nothing is returned. |

*Table 35. SQLBindFileToCol arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLINTEGER * | *IndicatorValue* | output (deferred) | Pointer to the location that contains an indicator value or array of values that indicate if the fetched LOB value is NULL. |

**Usage**

The LOB file reference arguments (file name, file name length, file reference options) refer to a file in the application's environment (on the client). Before fetching each row, the application must ensure that these variables contain the name of a file, the length of the file name, and a file option (create, overwrite, or append). These values can be changed between row fetch operations.

The application calls `SQLBindFileToCol()` once for each column that should be transferred directly to a file when a row is fetched. LOB data is written directly to the file without any data conversion, and without appending null-terminators.

*FileName*, *FileNameLength*, and *FileOptions* must be set before each fetch. When `SQLFetch()` or `SQLExtendedFetch()` is called, the data for any column that has been bound to a LOB file reference is written to the file or files that are pointed to by that file reference. Errors associated with the deferred input argument values of `SQLBindFileToCol()` are reported at fetch time. The LOB file reference, and the deferred *StringLength* and *IndicatorValue* output arguments are updated between fetch operations.

If `SQLExtendedFetch()` is used to retrieve multiple rows for the LOB column, *FileName*, *FileNameLength*, and *FileOptions* point to arrays of LOB file reference variables. In this case, *MaxFileNameLength* specifies the length of each element in the *FileName* array and is used by Db2 ODBC to determine the location of each element in the *FileName* array. The contents of the array of file references must be valid at the time of the `SQLExtendedFetch()` call. The *StringLength* and *IndicatorValue* pointers each point to an array whose elements are updated when `SQLExtendedFetch()` is called.

With `SQLExtendedFetch()`, multiple LOB values can be written to multiple files, or to the same file depending on the file names specified. If writing to the same file, the SQL_FILE_APPEND file option should be specified for each file name entry.

**Return codes**

After you call `SQLBindFileToCol()`, it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 36. SQLBindFileToCol SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**002 | Invalid column number. | The value specified for the argument *icol* was less than 1. |

*Table 36. SQLBindFileToCol SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**001 | Memory allocation failure. | Db2 ODBC is unable to allocate memory required to support execution or completion of the function. |
| **HY**009 | Invalid use of a null pointer. | *FileName* or *FileOptions* is a null pointer. |
| **HY**010 | Function sequence error. | The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.<br>The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC was unable to access memory required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The value specified for the argument *MaxFileNameLength* was less than 0. |
| **HY**C00 | Driver not capable. | The application is currently connected to a data source that does not support large objects. |

**Example**

```
/* Bind a file to a BLOB column */
rc = SQLBindFileToCol(hstmt,
                      1,
                      fileName,
                      &fileNameLength,
                      &fileOption,
                      14,
                      NULL,
                      &fileInd);
```

**Related reference**

SQLBindCol() - Bind a column to an application variable
`SQLBindCol()` binds a column to an application variable. You can call `SQLBindCol()` once for each column in a result set from which you want to retrieve data or LOB locators.

SQLExtendedFetch() - Fetch an array of rows
`SQLExtendedFetch()` extends the function of `SQLFetch()` by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that `SQLExtendedFetch()` returns.

SQLFetch() - Fetch the next row
`SQLFetch()` advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLBindFileToParam() - Bind a parameter marker to a file reference

SQLBindFileToParam() associates a parameter marker in an SQL statement to a file reference or to an array of file references. This association enables data from the file to be transferred directly into a LOB column when the statement is executed later.

### ODBC specifications for SQLBindFileToParam()

*Table 37. SQLBindFileToParam() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| No | No | No |

### Syntax

```
SQLRETURN SQLBindFileToParam (
             SQLHSTMT          StatementHandle,          /* hstmt */
             SQLUSMALLINT      TargetType,               /* ipar */
             SQLSMALLINT       DataType,                 /* fSqlType */
             SQLCHAR           *FileName,
             SQLSMALLINT       *FileNameLength,
             SQLUINTEGER       *FileOptions,
             SQLSMALLINT       MaxFileNameLength,
             SQLINTEGER        *IndicatorValue);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 38. SQLBindFileToParam arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *StatementHandle* | input | Statement handle. |
| SQLUSMALLINT | *TargetType* | input | Parameter marker number. Parameters are numbered sequentially, from left to right, starting at 1. |
| SQLSMALLINT | *DataType* | input | SQL data type of the column. The data type must be one of these types:<br>• SQL_BLOB<br>• SQL_CLOB<br>• SQL_DBCLOB |
| SQLCHAR * | *FileName* | input (deferred) | Pointer to the location that contains the file name or an array of file names when the statement (*StatementHandle*) is executed. This is the absolute path name of the file.<br>This argument cannot be NULL. |

*Table 38. SQLBindFileToParam arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLSMALLINT * | *FileNameLength* | input (deferred) | Pointer to the location that contains the length of the file name (or an array of lengths) at the time of the next `SQLExec()` or `SQLExecDirect()` using *StatementHandle*. If this pointer is NULL, ODBC treats *FileName* as a null-terminated string. The result is the same as if a length of SQL_NTS is passed.<br><br>The maximum value of the file name length is 255. |
| SQLUINTEGER * | *FileOptions* | input (deferred) | Pointer to the location that contains the file option (or an array of file options) to be used when the file is read. The location is accessed when the statement (*StatementHandle*) is executed. Only one option is supported, and it must be specified:<br><br>**SQL_FILE_READ**<br>A regular file that can be opened, read, and closed. The length is computed when the file is opened.<br><br>This pointer cannot be NULL. |
| SQLSMALLINT | *MaxFileNameLength* | input | The length of the *FileName* buffer. If the application calls `SQLSetStmtAttr()` to specify multiple values for each parameter, this is the length of each element in the *FileName* array. |
| SQLINTEGER * | *IndicatorValue* | input (deferred) | The pointer to the location that contains an indicator value or array of values, which are set to SQL_NULL_DATA if the LOB data value is to be null. The value at the location must be set to 0 or the pointer must be set to null when the data value is not null. |

**Usage**

The LOB file reference arguments (file name, file name length, file reference options) refer to a file within the application's environment (on the client). Before calling `SQLExecute()` or `SQLExecDirect()`, the application must ensure that this information is available in the deferred input buffers. These values can be changed between `SQLExecute()` calls.

The application calls `SQLBindFileToParam()` once for each parameter marker whose valueis obtained directly from a file when a statement is executed. Before the statement is executed, the *FileName*, *FileNameLength*, and *FileOptions* values must be set. When the statement is executed, the data for any parameter that has been bound using `SQLBindFileToParam()` is read from the referenced file and passed to the server.

If the application uses `SQLSetStmtAttr()` to specify multiple values for each parameter, *FileName, FileNameLength,* and *FileOptions* point to an array of LOB file reference variables. In this case, *MaxFileNameLength* specifies the length of each element in the *FileName* array and is used by Db2 ODBC to determine the location of each element in the *FileName* array.

A LOB parameter marker can be associated with an input file using `SQLBindFileToParam()`, or with a stored buffer using `SQLBindParameter()`. The most recent bind parameter function call determines the type of binding that is in effect.

### Return codes

After you call SQLBindFileToParam(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 39. SQLBindFileToParam SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is unable to allocate the memory that is required to support execution or completion of the function. |
| **HY**004 | SQL data type out of range. | The value specified for *DataType* was not a valid SQL type for this function call. |
| **HY**009 | Invalid argument value. | *FileName* or *FileOptions* is a null pointer. |
| **HY**010 | Function sequence error. | The function was called while in a data-at-execute (SQLParamdata(), SQLPutData()) operation.<br>The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC was unable to access memory that is required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The value specified for the input argument *MaxFileNameLength* was less than 0. |
| **HY**093 | Invalid parameter number. | The value specified for *TargetType* was less than 1. |
| **HY**C00 | Driver not capable. | Db2 ODBC does not support "catalog" as a qualifier for table name. |

### Example

```
/* Bind a file reference to a parameter */
rc = SQLBindFileToParam(hstmt,
                        3,
                        SQL_BLOB,
                        fileName,
                        &fileNameLength,
                        &fileOption,
                        14,
                        &fileInd);
```

### Related reference

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

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

SQLBindParameter() binds parameter markers to application variables and extends the capability of the SQLSetParam() function.

### ODBC specifications for SQLBindParameter()

*Table 40. SQLBindParameter() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 2.0 | No | No |

### Syntax

For 31-bit applications, use the following 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);
```

For 64-bit applications, use the following syntax:

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

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 41. 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 41. SQLBindParameter() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLSMALLINT | *fParamType* | input | Specifies the type of parameter. You can specify the following types of parameters:<br>• 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.<br>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.<br>• SQL_PARAM_INPUT_OUTPUT: The parameter marker is associated with an input/output parameter of the called stored procedure.<br>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.<br>• 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.<br>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. |

*Table 41. SQLBindParameter() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLSMALLINT | *fCType* | input | Specifies the C data type of the parameter. The following types are supported:<br>• SQL_C_BIGINT<br>• SQL_C_BINARY<br>• SQL_C_BINARYXML<br>• SQL_C_BIT<br>• SQL_C_BLOB_LOCATOR<br>• SQL_C_CHAR<br>• SQL_C_CLOB_LOCATOR<br>• SQL_C_DBCHAR<br>• SQL_C_DBCLOB_LOCATOR<br>• SQL_C_DECIMAL64<br>• SQL_C_DECIMAL128<br>• SQL_C_DOUBLE<br>• SQL_C_FLOAT<br>• SQL_C_LONG<br>• SQL_C_SHORT<br>• SQL_C_TYPE_DATE<br>• SQL_C_TYPE_TIME<br>• SQL_C_TYPE_TIMESTAMP<br>• SQL_C_TYPE_TIMESTAMP_EXT<br>• SQL_C_TINYINT<br>• SQL_C_WCHAR<br>Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type to the type indicated in *fSqlType*. |

*Table 41. SQLBindParameter() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLSMALLINT | *fSqlType* | input | Specifies the SQL data type of the parameter. The supported types are:<br>• SQL_BIGINT<br>• SQL_BINARY<br>• SQL_BLOB<br>• SQL_BLOB_LOCATOR<br>• SQL_CHAR<br>• SQL_CLOB<br>• SQL_CLOB_LOCATOR<br>• SQL_DBCLOB<br>• SQL_DBCLOB_LOCATOR<br>• SQL_DECFLOAT<br>• SQL_DECIMAL<br>• SQL_DOUBLE<br>• SQL_FLOAT<br>• SQL_GRAPHIC<br>• SQL_INTEGER<br>• SQL_LONGVARBINARY<br>• SQL_LONGVARCHAR<br>• SQL_LONGVARGRAPHIC<br>• SQL_NUMERIC<br>• SQL_REAL<br>• SQL_ROWID<br>• SQL_SMALLINT<br>• SQL_TYPE_DATE<br>• SQL_TYPE_TIME<br>• SQL_TYPE_TIMESTAMP<br>• SQL_VARBINARY<br>• SQL_VARCHAR<br>• SQL_VARGRAPHIC<br>• SQL_XML<br>**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. |

*Table 41. SQLBindParameter() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLUINTEGER (31-bit) or SQLULEN (64-bit) [1] | *cbColDef* | input | Specifies the precision of the corresponding parameter marker. The meaning of this precision depends on what data type the *fSqlType* argument denotes:<br>• 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.<br>• For a double-byte character string (for example, SQL_GRAPHIC), this is the maximum length in double-byte characters for this parameter.<br>• For SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision.<br>• For SQL_DECFLOAT, the *cbColDef* argument must specify the precision of the parameter marker, which is 16 if the column is DECFLOAT(16) or 34 if the column is DECFLOAT(34).<br>• For SQL_ROWID, this must be set to 40, the maximum length in bytes for this data type. Otherwise, an error is returned.<br>• 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).<br>Other than the values for the *fSqlType* argument that are mentioned here, the *ibScale* argument is ignored. |

*Table 41. 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:<br>• As an input argument (when the *fParamType* argument specifies SQL_PARAM_INPUT, or SQL_PARAM_INPUT_OUTPUT), *rgbValue* exhibits the following behavior:<br>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.<br>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.<br>If SQLSetStmtAttr() is called to specify multiple values for the parameter, then *rgbValue* is a pointer to an input buffer array of *cbValueMax* bytes.<br>• 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.<br>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. |

*Table 41. SQLBindParameter() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLINTEGER (31-bit) or SQLLEN (64-bit) [2] | *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.<br><br>If this buffer is an array, the value specifies the size of each element in that array. Call SQLSetStmtAttr() to specify multiple values for each parameter. For non-character and non-binary data, this argument is ignored.<br><br>This length is assumed to be the length that is associated with the C data type in these cases.<br><br>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:<br><br>• 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).<br>• 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 41. SQLBindParameter() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLINTEGER * (31-bit) SQLLEN * (64-bit)[2] | *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:<br>• 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.<br>To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA.<br>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.<br>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.<br>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.<br>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. |

*Table 41. SQLBindParameter() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| Continued SQLINTEGER * (31-bit) SQLLEN * (64-bit)[2] | *pcbValue* | input (deferred), output (deferred), or input (deferred) and output (deferred) | • If you use SQLSetStmtAttr() 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.<br>If you use SQLBindParameter(), you can specify values for SQL_UNASSIGNED and SQL_DEFAULT_PARAM in the *pcbValue* argument. These values require that you enable extended indicator support with the INI keyword EXTENDEDINDICATOR or the SQL_ATTR_EXTENDED_INDICATORS connection variable. If you specify SQL_UNASSIGNED or SQL_DEFAULT PARAM when extended indicator support is disabled, the results are the same as specifying SQL_NULL_DATA.<br>**SQL_DEFAULT_PARAM**<br>The target column of the bound parameter is set to its defined DEFAULT value.<br>**SQL_UNASSIGNED**<br>The target column of the bound parameter is ignored for UPDATE, and MERGE UPDATE operations. The parameter is handled the same way as the DEFAULT keyword for INSERT, and MERGE INSERT operations.<br>• 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:<br>– number of bytes available to return in *rgbValue*, excluding the nul-termination character.<br>– SQL_NULL_DATA<br>– SQL_NO_TOTAL if the number of bytes available to return cannot be determined. |

**Notes:**

1. For 64-bit applications, the data type SQLUINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLULEN is recommended.
2. For 64-bit applications, the data type SQLINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLLEN is recommended.

### Usage

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 database management system 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.
- 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.

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 SQLSetStmtAttr() 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.

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 `SQLSetStmtAttr()` 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

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;
```

**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 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, the CCSID that is specified for the CURRENTAPPENSCH keyword is an EBCDIC CCSID, 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 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, or the CCSID that is specified for CURRENTAPPENSCH is a Unicode CCSID (1200, 1208, 13488 or 17584).
  - 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, or the CCSID that is specified for CURRENTAPPENSCH is an ASCII CCSID.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLBindParameter() call.

### Return codes

After you call SQLBindParameter(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 42. SQLBindParameter() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **07**006 | 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.) |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | An unrecoverable system error occurs. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**003 | Program type out of range. | The *fCType* argument is not a valid data type or SQL_C_DEFAULT. |
| **HY**004 | Invalid SQL data type. | The specified value for the *fSqlType* argument is not a valid SQL data type. |
| **HY**009 | Invalid use of a null pointer. | The argument *OutputHandlePtr* is a null pointer. |
| **HY**010 | 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. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The specified value for the *cbValueMax* argument is less than 0. |
| **HY**093 | Invalid parameter number. | The specified value for the *ipar* argument is less than 1. |
| **HY**094 | Invalid scale value. | HY094 is returned when the specified value for the *fSqlType* is SQL_TYPE_TIMESTAMP and the value for the *ibScale* argument is less than 0 or greater than 12. |
| **HY**104 | Invalid precision value. | This SQLSTATE is returned because 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.<br>This SQLSTATE is returned for the following reason:<br>• The specified value for the *fCType* argument is SQL_C_TYPE_TIMESTAMP_EXT, 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 12. |
| **HY**105 | Invalid parameter type. | The *fParamType* argument does not specify one of the following values:<br>• SQL_PARAM_INPUT<br>• SQL_PARAM_OUTPUT<br>• SQL_PARAM_INPUT_OUTPUT |

*Table 42. SQLBindParameter() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**C00 | Driver not capable. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• 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**

The following example shows an application that binds a variety of data types to a set of parameters.

```
/* ... */
    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;
/* ... */
```

```
    /* 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
/* ... */
```

*Figure 9. An application that binds data types to parameters*

**Related concepts**

Using arrays to pass parameter values
Db2 ODBC provides an array input method for updating Db2 tables.

Example of binding UTF-8 data to parameter markers
You can use SQLBindParameter() to bind application variables that contain UTF-8 data to INTEGER, CHAR, and GRAPHIC parameter markers.

Data types and data conversion
When you write a Db2 ODBC application, you must work with both SQL data types and C data types. The database server uses SQL data types, and the application uses C data types.

Application encoding schemes and Db2 ODBC
Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The Db2 ODBC driver supports input and output character string arguments to ODBC APIs and input and output host variable data in each of these encoding schemes.

**Related reference**

SQLExecDirect() - Execute a statement directly
SQLExecDirect() prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
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.

SQLParamData() - Get next parameter for which a data value is needed
SQLParamData() is used in conjunction with SQLPutData() to send long data in pieces. You can also use this function to send fixed-length data.

SQLPutData() - Pass a data value for a parameter
SQLPutData() supplies 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. This result set is retrieved by the same functions that process a result set that is generated by a query.

SQLSetStmtAttr() - Set statement attributes

SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

## SQLBulkOperations() - Add, update, delete or fetch a set of rows

SQLBulkOperations() adds new rows to the base table or view that is associated with a dynamic cursor for the current query.

### ODBC specifications for SQLBulkOperations()

*Table 43. SQLBulkOperations() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

```
SQLRETURN SQLBulkOperations (
            SQLHSTMT      StatementHandle,
            SQLSMALLINT   Operation);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 44. SQLBulkOperations arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *StatementHandle* | Input | Statement handle. |
| SQLSMALLINT | *Operation* | Input | Operation to perform: SQL_ADD.<br>Db2 ODBC does not support the following operations:<br>• SQL_UPDATE_BY_BOOKMARK<br>• SQL_DELETE_BY_BOOKMARK<br>• SQL_FETCH_BY_BOOKMARK |

### Usage

Before calling SQLBulkOperations(), you need to ensure that the required bulk operation is supported. To check for support, call SQLGetInfo() with an *InfoType* of SQL_DYNAMIC_CURSOR_ATTRIBUTES1 or SQL_DYNAMIC_CURSOR_ATTRIBUTES2. Check the following attributes to verify that support is available:

- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_BULK_DELETE_BY_BOOKMARK
- SQL_CA1_BULK_FETCH_BY_BOOKMARK

A column can be ignored when bulk operations are performed with SQLBulkOperations(). To ignore a column, call SQLBindCol(), and set the column length and indicator buffer (*pcbValue*) to SQL_COLUMN_IGNORE.

After a call to SQLBulkOperations():

- The buffer to which the SQL_ATTR_ROWS_FETCHED_PTR statement attribute points contains the number of rows that are affected by the call.
- The row status array, to which the SQL_ATTR_ROW_STATUS_PTR statement attribute points, contains the result of the operation.
- The block cursor position is undefined. The application must call SQLFetchScroll() to set the cursor position. The application needs to call SQLFetchScroll() with a *FetchOrientation* argument of SQL_FETCH_FIRST, SQL_FETCH_LAST, or SQL_FETCH_ABSOLUTE. The cursor position is undefined if the application calls SQLFetch(), or calls SQLFetchScroll() with a *FetchOrientation* argument of SQL_FETCH_PRIOR, SQL_FETCH_NEXT, or SQL_FETCH_RELATIVE.

The application does *not* need to:

- Call SQLFetch() or SQLFetchScroll() before calling SQLBulkOperations().
- Set the SQL_ATTR_ROW_OPERATION_PTR statement attribute for SQLBulkOperations() calls. Rows cannot be ignored when bulk operations are performed with SQLBulkOperations().

When the *Operation* argument is SQL_ADD, and the select list of the query that is associated with the cursor contains more than one reference to the same column, an error is generated.

**Row status array:** The row status array contains status values for each row of data in the rowset after a call to SQLBulkOperations(). This array is initially populated by a call to SQLBulkOperations() if SQLFetch() or SQLFetchScroll() has not been called before SQLBulkOperations() is called. The SQL_ATTR_ROW_STATUS_PTR statement attribute points to the row status array. The number of elements in the row status array should equal the number of rows in the rowset, as defined by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.

#### Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

#### Diagnostics

*Table 45. SQLBulkOperations SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (Function returns SQL_SUCCESS_WITH_INFO.) |
| **07**006 | Invalid conversion. | The *Operation* argument was SQL_ADD, and the data value in the application buffers could not be converted to the data type of a column in the result set. |
| **22**001 | String data right truncation. | The assignment of a character or binary value to a column resulted in the truncation of non-blank (for characters) or non-null (for binary) characters or bytes. |
| **22**003 | Numeric value out of range. | The *Operation* argument was SQL_ADD. The assignment of a numeric value to a column in the result set caused the whole (as opposed to fractional) part of the number to be truncated. |

*Table 45. SQLBulkOperations SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **22**008 | Invalid datetime format or datetime field overflow. | One of the following conditions occurred:<br>• The *Operation* argument was SQL_ADD. The assignment of a date or timestamp value to a column in a result set caused the year, month, or day field to be out of range.<br>• The *Operation* argument was SQL_ADD. A datetime arithmetic operation on data that was sent to a column in the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result that was outside the permissible range of values for the field, or was invalid based on the natural rules for datetime values for the Gregorian calendar. |
| **22**018 | Error in assignment. | The argument *Operation* was SQL_ADD. The data value that was assigned to a column was incompatible with the data type of the associated column in the result set. |
| **23**000 | Integrity constraint violation. | An integrity constraint was violated. One of the following conditions occurred:<br>• The *Operation* argument was SQL_ADD. A column that was not bound is defined as NOT NULL and has no default.<br>• The *Operation* argument was SQL_ADD. The length that was specified in the bound *pcbValue* was SQL_COLUMN_IGNORE, and the column did not have a default value. |
| **24**000 | Invalid cursor state. | The *StatementHandle* was in an executed state, but no result set was associated with the *StatementHandle*. |
| **40**001 | Transaction rollback. | The transaction in which the fetch was executed was terminated to prevent deadlock. |
| **40**003 | Statement completion unknown. | The associated connection failed during the execution of this function. The state of the transaction cannot be determined. |
| **42***xxx*[1] | Syntax error or access rule violation. | These SQLSTATEs indicate one of the following errors:<br>• For **42**5*xx*, the authorization ID does not have permission to perform the operation that was requested in the *Operation* argument.<br>• For **42***xxx*, a variety of syntax or access problems with the statement occur. |
| **44**000 | WITH CHECK OPTION violation. | The *Operation* argument was SQL_ADD. An insert or update was performed on a viewed table or a table that was derived from the viewed table. The viewed table was created by specifying WITH CHECK OPTION. One or more rows that were affected by the insert are no longer present in the viewed table. |
| **HY**000 | General error. | An error occurred for which there was no specific SQLSTATE. The error message that was returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause. |

*Table 45. SQLBulkOperations SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**001 | Memory allocation error. | Db2 ODBC was unable to allocate memory required to support execution or completion of the function. Process-level memory might have been exhausted for the application process. Consult the operating system configuration for information on process-level memory limitations. |
| **HY**010 | Function sequence error. | The function was called while in a data-at-execute (`SQLParamData()` or `SQLPutData()`) operation. |
| **HY**011 | Operation invalid at this time. | The SQL_ATTR_ROW_STATUS_PTR statement attribute was set between calls to `SQLFetch()` or `SQLFetchScroll()`, and `SQLBulkOperations`. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC was unable to access memory that was required to support execution or completion of this function. |
| **HY**090 | Invalid string or buffer length. | One of the following conditions occurred:<br>• The *Operation* argument was SQL_ADD. A data value was a null pointer, and the column length value was not 0, SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.<br>• The *Operation* argument was SQL_ADD. A data value was not a null pointer. The C data type was SQL_C_BINARY or SQL_C_CHAR. The column length value was less than 0, but not equal to SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NTS, or SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. |
| **HY**092 | Invalid attribute identifier. | The *Operation* argument was SQL_ADD. The SQL_ATTR_CONCURRENCY statement attribute was set to SQL_CONCUR_READ_ONLY. |
| **HY**C00 | Driver not capable. | Db2 ODBC or the data source does not support the operation that was requested in the *Operation* argument. |

**Note:**

1. *xxx* refers to any SQLSTATE with that class code. For example, **37***xxx* refers to any SQLSTATE with class code '**37**'.

**Related concepts**

The ODBC row status array
The row status array returns the status of each row in the rowset.

**Related tasks**

Performing bulk inserts with SQLBulkOperations()
You can insert new rows into a table or view at a data source with a call to `SQLBulkOperations()`.

Providing long data for bulk inserts and positioned updates
To provide long data for bulk inserts or positioned updates, use `SQLBulkOperations()` or `SQLSetPos()` calls.

**Related reference**

SQLBindCol() - Bind a column to an application variable

SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLFetchScroll() - Fetch the next row
SQLFetchScroll() fetches the specified rowset of data from the result set of a query and returns data for all bound columns. Rowsets can be specified at an absolute position or a relative position.

## SQLCancel() - Cancel statement

SQLCancel() terminates an SQLExecDirect() or SQLExecute() sequence prematurely.

### ODBC specifications for SQLCancel()

*Table 46. SQLCancel() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

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

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 47. SQLCancel() arguments*

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

### Usage

Use SQLCancel() to cancel the following types of processing on a statement:

- Data-at-execution sequences on the current thread.
- Functions running on the statement on another thread.

### Canceling a data-at-execution sequence

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; instead, use SQLCloseCursor() to close cursors.

### Canceling functions in multithreaded applications

When you execute a multithreaded application, you can cancel a function that is running synchronously on another thread. To cancel the function, you must call `SQLCancel()` on a different thread with the same statement handle as that used by the target function, and set the `INTERRUPT` keyword in the ODBC initialization file to either `INTERRUPT=1` (the default setting) or `INTERRUPT=2`. In Db2 ODBC, `INTERRUPT=1` and `INTERRUPT=2` exhibit the same behavior, which is set to always drop the connection on a `SQLCancel()`. After you call `SQLCancel()`, Db2 ODBC sets the return code to either SQL_SUCCESS or SQL_ERROR (no SQLSTATE) to indicate whether the cancel request was processed successfully. If the request was successful, the connection associated with the statement handle is dropped and the canceled function returns SQLCODE -924 and SQLSTATE 58006. In order for the statement handle to process additional database requests, you must establish a new connection with the database server.

If an SQL statement is being executed when `SQLCancel()` is called on another thread to cancel the statement execution, it is possible for the execution to succeed and return SQL_SUCCESS while the cancel is also successful. In this case, the connection associated with the statement is dropped regardless of the return code, so you will not be able to process additional database requests on that statement until you re-establish the connection.

### Return codes

After you call `SQLCancel()`, it returns one of the following values:

- SQL_SUCCESS
- SQL_INVALID_HANDLE
- SQL_ERROR

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 48. `SQLCancel()` SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**013 | 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 concepts

Input and retrieval of long data in pieces
When an application must manipulate long data values, loading the entire values into storage can become impractical. For this reason, Db2 ODBC provides a technique that enables you to handle long data values in pieces.

### Related reference

SQLParamData() - Get next parameter for which a data value is needed
`SQLParamData()` is used in conjunction with `SQLPutData()` to send long data in pieces. You can also use this function to send fixed-length data.

SQLPutData() - Pass a data value for a parameter

SQLPutData() supplies 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. This result set is retrieved by the same functions that process a result set that is generated by a query.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

**Related information**
-924 (Db2 Codes)

# SQLCloseCursor() - Close a cursor and discard pending results

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

### ODBC specifications for SQLCloseCursor()

*Table 49. SQLCloseCursor() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

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

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 50. 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 51. SQLCloseCursor() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (SQLCloseCursor() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **24**000 | Invalid cursor state. | No cursor is open on the statement handle. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is unable to allocate memory that is required execute or complete the function. |
| **HY**010 | 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. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is unable to access memory that is required to support execution or completion of the function. |

### Example

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

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

### Related reference

SQLGetConnectAttr() - Get current attribute setting
SQLGetConnectAttr() returns the current setting of a connection attribute and also allows you to set these attributes.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetConnectAttr() - Set connection attributes
SQLSetConnectAttr() sets attributes that govern aspects of connections.

SQLSetStmtAttr() - Set statement attributes

SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

# SQLColAttribute() - Get column attributes

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.

### ODBC specifications for SQLColAttribute()

*Table 52. SQLColAttribute() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

```
SQLRETURN  SQLColAttribute  (SQLHSTMT         StatementHandle,
                            SQLSMALLINT      ColumnNumber,
                            SQLSMALLINT      FieldIdentifier,
                            SQLPOINTER       CharacterAttributePtr,
                            SQLSMALLINT      BufferLength,
                            SQLSMALLINT      *StringLengthPtr,
                            SQLPOINTER       NumericAttributePtr);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 53. 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. |
| SQLSMALLINT | *FieldIdentifier* | input | The field in row *ColumnNumber* that is to be returned. See Table 54 on page 130. |
| 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. |

*Table 53. SQLColAttribute() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLSMALLINT * | *StringLengthPtr* | output | Pointer to a buffer in which to return the total number of bytes (excluding the nul-termination character) that are available to return in **CharacterAttributePtr*.<br><br>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.<br><br>For all other types of data, the value of *BufferLength* is ignored, and Db2 ODBC assumes that the size of **CharacterAttributePtr* is 32 bits. |
| SQLPOINTER | *NumericAttributePtr* | output | Pointer to an integer buffer in which to return the value in the *FieldIdentifier* field of the *ColumnNumber* row, 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 ANSI/ISO SQL standard of 1992 and database management system 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.

The following table 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 54. 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. |

*Table 54. SQLColAttribute() field identifiers (continued)*

| Field identifier | Information returned in arguments | Description |
|---|---|---|
| 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 and for the XML data type. |
| 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. |
| 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. |
| 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.<br><br>**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.<br><br>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. |

*Table 54. SQLColAttribute() field identifiers (continued)*

| Field identifier | Information returned in arguments | Description |
|---|---|---|
| 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. This value is 0 for the XML data type. |
| 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 suffix is not applicable to this data type. |
| 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.<br><br>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. |

*Table 54. SQLColAttribute() field identifiers (continued)*

| Field identifier | Information returned in arguments | Description |
|---|---|---|
| SQL_DESC_NUM_PREC_RADIX | *NumericAttributePtr* | The precision of each digit in a numeric value. The following values are commonly returned:<br>• 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.<br>• 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.<br>• This field is set to 0 for all nonnumeric data types. |
| 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.<br>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.<br>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. |

*Table 54. SQLColAttribute( ) field identifiers (continued)*

| Field identifier | Information returned in arguments | Description |
|---|---|---|
| SQL_DESC_PRECISION (SQL_COLUMN_PRECISION)[1] | *NumericAttributePtr* | The precision in units of digits if the column is:<br>• SQL_BIGINT<br>• SQL_DECIMAL<br>• SQL_NUMERIC<br>• SQL_DOUBLE<br>• SQL_FLOAT<br>• SQL_DECFLOAT<br>• SQL_INTEGER<br>• SQL_REAL<br>• SQL_SMALLINT<br>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.<br>If the column is a graphic SQL data type, the precision indicates the maximum number of double-byte characters that the column can hold.<br>If the column is the XML data type, the precision is 0. |
| SQL_DESC_SCALE (SQL_COLUMN_SCALE)[1] | *NumericAttributePtr* | The scale attribute of the column. |
| 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. |
| SQL_DESC_SEARCHABLE (SQL_COLUMN_SEARCHABLE)[1] | *NumericAttributePtr* | Indicates if the column data type is searchable:<br>• SQL_PRED_NONE (SQL_UNSEARCHABLE in ODBC 2.0) if the column cannot be used in a WHERE clause.<br>• 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.<br>• 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.<br>• SQL_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator. |

*Table 54. SQLColAttribute() field identifiers (continued)*

| Field identifier | Information returned in arguments | Description |
|---|---|---|
| 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. 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). |
| 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. |
| SQL_DESC_UPDATABLE (SQL_COLUMN_UPDATABLE)[1] | *NumericAttributePtr* | Indicates if the column data type is a data type that can be updated:<br>• SQL_ATTR_READWRITE_UNKNOWN is returned in *NumericAttributePtr* for all Db2 SQL data types.<br>• 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:<br>– SQL_DESC_UPDATABLE<br>– 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 55. SQLColAttribute() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (SQLColAttribute() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **01**004 | 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.) |
| **07**005 | 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. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate memory that is required to support execution or completion of the function. |
| **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. |
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• The function is called prior to SQLPrepare() or SQLExecDirect() for the statement handle that the *StatementHandle* argument specifies.<br>• 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. |
| **HY**090 | Invalid string or buffer length. | The value that is specified for the *BufferLength* argument is less than 0. |
| **HY**091 | 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. |
| **HY**C00 | 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**

Refer to SQLColAttribute() for a related example. In this example, SQLColAttribute() retrieves the display length for a column.

**Related reference**

C and SQL data types
Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

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.

Length of SQL data types
The length of a column is the maximum number of bytes that are returned to the application when data is transferred to its default C data type.

Precision of SQL data types
The precision of a numeric column or parameter refers to the maximum number of digits that are used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter.

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.

SQLBindCol() - Bind a column to an application variable
SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLDescribeCol() - Describe column attributes
SQLDescribeCol() returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

SQLExtendedFetch() - Fetch an array of rows
SQLExtendedFetch() extends the function of SQLFetch() by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLColAttributes() - Get column attributes

SQLColAttributes() is a deprecated function and is replaced by SQLColAttribute().

**ODBC specifications for SQLColAttributes()**

*Table 56. SQLColAttributes() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | No | No |

### Syntax

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

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 57. 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 the SQLColAttribute() function description. |
| 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. |

### Related reference

SQLColAttribute() - Get column attributes
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.

## SQLColumnPrivileges() - Get column privileges

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

### ODBC specifications for SQLColumnPrivileges()

*Table 58. SQLColumnPrivileges() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### 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 59 on page 139 lists the data type, use, and description for each argument in this function.

*Table 59. 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 | *cbSchemaNam e* | 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 60 on page 140. 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 ANSI/ISO SQL standard of 1992 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 database management system supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected database management system supports

- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected database management system supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected database management system supports

Note that the *szColumnName* argument accepts a search pattern.

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. The following table lists the columns in the result set that `SQLColumnPrivileges()` currently returns.

*Table 60. 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. |
| 7 | PRIVILEGE | VARCHAR(128) | Indicates the column privilege. This can be:<br>• ALTER<br>• CONTROL<br>• DELETE<br>• INDEX<br>• INSERT<br>• REFERENCES<br>• SELECT<br>• UPDATE<br>Supported privileges are based on the data source to which you are connected.<br>Most IBM relational database management systems do not offer column-level privileges at the column level. Db2 for z/OS and DB2 for VSE & VM support the UPDATE column privilege; each updatable column receives one row in this result set. For all other privileges for Db2 for z/OS and DB2 for VSE & VM, and for all privileges for other IBM relational database management systems, 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.<br>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

### Diagnostics

Table 61 on page 141 lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 61. SQLColumnPrivileges() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is open on the statement handle. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**009 | Invalid use of a null pointer. | The *szTableName* argument is null. |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |
| **HY**090 | Invalid string or buffer length. | The value of one of the name length arguments is less than 0 and not equal to SQL_NTS. |
| **HY**C00 | Driver not capable. | Db2 ODBC does not support "catalog" as a qualifier for table name. |

### Example

The following example 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
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        sprintf(cur_name, " Column:
        if (strcmp(cur_name, pre_name) != 0) {
            printf("\nname);
            printf("   Grantor         Grantee         Privilege  Grantable\n");
            printf("   --------------- --------------- ---------- ---\n");
        }
        strcpy(pre_name, cur_name);
        printf("
        printf("
        printf("
        printf(" grantable.s);
    }                           /* endwhile */
/* ... */
```

*Figure 10. An application that retrieves user privileges on table columns*

**Related concepts**

Input arguments on catalog functions

Input arguments identify or constrain the amount of information that a catalog function returns.

**Related reference**

SQLColumns() - Get column information

`SQLColumns()` returns a list of columns in the specified tables. The information is returned in an SQL result set, which can be retrieved by using the same functions that fetch a result set that a query generates.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLTables() - Get table information

`SQLTables()` returns a list of table names and associated information that is stored in the system catalog of the connected data source. The list of table names is returned as a result set. You can retrieve this result set with the same functions that process a result set generated by a query.

# SQLColumns() - Get column information

`SQLColumns()` returns a list of columns in the specified tables. The information is returned in an SQL result set, which can be retrieved by using the same functions that fetch a result set that a query generates.

**ODBC specifications for SQLColumns()**

*Table 62. SQLColumns() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | No |

### 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

The following table lists the data type, use, and description for each argument in this function.

*Table 63. 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.<br>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 64 on page 144 lists the columns in the result set.

The *szSchemaName, szTableName*, and *szColumnName* arguments accept search patterns.

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 ANSI/ISO SQL standard of 1992 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 database management system supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected database management system supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected database management system supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected database management system 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. The following table lists the columns in the result set that `SQLColumns()` currently returns.

*Table 64. 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. |
| 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. |
| 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.<br><br>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.<br><br>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.<br><br>For the XML data type, the length of zero is returned. |

*Table 64. Columns returned by SQLColumns() (continued)*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 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. |
| 9 | DECIMAL_DIGITS | SMALLINT | Indicates the scale of the column. NULL is returned for data types where scale is not applicable. |
| 10 | NUM_PREC_RADIX | SMALLINT | Specifies 10, 2, or NULL.<br>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.<br>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.<br>For numeric data types, the database management system can return a NUM_PREC_RADIX value of either 10 or 2.<br>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.<br>Contains SQL_NULLABLE if the column accepts null values. |
| 12 | REMARKS | VARCHAR(762) | Contains any descriptive information about the column. |

*Table 64. Columns returned by SQLColumns() (continued)*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 13 | COLUMN_DEF | VARCHAR(254) | Identifies the default value for the column.<br>If the default value is a numeric literal, this column contains the character representation of the numeric literal with no enclosing single quotes.<br>If the default value is a character string, this column is that string, enclosed in single quotes.<br>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.<br>If NULL was specified as the default value, this column returns the word NULL, with no enclosing single quotes.<br>If the default value cannot be represented without truncation, this column contains the value TRUNCATED with no enclosing single quotes.<br>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.<br>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_DATETIME_SUB | SMALLINT | The subtype code for datetime data types can be one of the following values:<br>• SQL_CODE_DATE<br>• SQL_CODE_TIME<br>• SQL_CODE_TIMESTAMP<br>For all other data types, this column returns NULL. |

*Table 64. Columns returned by SQLColumns() (continued)*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 16 | CHAR_OCTET_LENGTH | INTEGER | Contains the maximum length in bytes for a character data column. For single-byte character sets, this is the same as COLUMN_SIZE. For the XML type, zero is returned. For data types other than character data types or XML data type, 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 the preceding table 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 for z/OS server should expect the result set columns that are listed in Table 64 on page 144.

### Return codes

After you call SQLColumns(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 65. SQLColumns() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is open on the statement handle. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |

*Table 65. SQLColumns() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**090 | Invalid string or buffer length. | The value of one of the name length argument is less than 0 and not equal to SQL_NTS. |
| **HY**C00 | Driver not capable. | Db2 ODBC does not support "catalog" as a qualifier for table name. |

**Example**

The following example 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, "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:
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("   name.s);
        if (nullable == SQL_NULLABLE) {
            printf(", NULLABLE");
        } else {
            printf(", NOT NULLABLE");
        }
        printf(", name.s);
        if (length_ind != SQL_NULL_DATA) {
            printf(" (
        } else {
            printf("(\n");
        }
        if (scale_ind != SQL_NULL_DATA) {
            printf(",
        } else {
            printf(")\n");
        }
    }                           /* endwhile */
/* ... */
```

*Figure 11. An application that returns information about table columns*

**Related concepts**

Input arguments on catalog functions
Input arguments identify or constrain the amount of information that a catalog function returns.

**Related reference**

C and SQL data types
Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

Length of SQL data types

The length of a column is the maximum number of bytes that are returned to the application when data is transferred to its default C data type.

Precision of SQL data types
The precision of a numeric column or parameter refers to the maximum number of digits that are used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter.

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.

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

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSpecialColumns() - Get special (row identifier) columns
`SQLSpecialColumns()` returns unique row identifier information (primary key or unique index) for a table. The information is returned in an SQL result set. You can retrieve this result set with the same functions that process a result set that is generated by a query.

SQLTables() - Get table information
`SQLTables()` returns a list of table names and associated information that is stored in the system catalog of the connected data source. The list of table names is returned as a result set. You can retrieve this result set with the same functions that process a result set generated by a query.

# SQLConnect() - Connect to a data source

`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 can call `SQLConnect()`. Subsequently, you must call `SQLConnect()` before you allocate a statement handle.

### ODBC specifications for SQLConnect()

*Table 66. SQLConnect() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLConnect       (SQLHDBC           hdbc,
                              SQLCHAR     FAR   *szDSN,
                              SQLSMALLINT       cbDSN,
                              SQLCHAR     FAR   *szUID,
                              SQLSMALLINT       cbUID,
                              SQLCHAR     FAR   *szAuthStr,
                              SQLSMALLINT       cbAuthStr);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 67. 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 relational database management systems 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 68. SQLConnect() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**001 | Unable to connect to data source. | This SQLSTATE is returned for one or more of the following reasons:<br>• Db2 ODBC is not able to establish a connection with the data source.<br>• The connection request is rejected because a connection that was established with embedded SQL already exists. |
| **08**002 | Connection in use. | The specified connection handle is being used to establish a connection with a data source, and that connection is still open. |
| **08**004 | The application server rejected establishment of the connection. | This SQLSTATE is returned for one or more of the following reasons:<br>• The data source rejects the establishment of the connection.<br>• The number of connections that are specified by the MAXCONN keyword has been reached. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |

*Table 68. SQLConnect() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**024 | Invalid argument value. | A nonmatching double quotation mark (") is found in the *szDSN*, *szUID*, or *szAuthStr* arguments. |
| **HY**090 | Invalid string or buffer length. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• 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.<br>• 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. |
| **HY**501 | 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 relational database management systems is not supported. SQLConnect() must be called before any SQL statements can be executed.

### Example

The following example 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:
        SQLDisconnect(*hdbc);
        SQLFreeHandle (SQL_HANDLE_DBC, *hdbc);
        return (SQL_ERROR);
    } else {                    /* Print connection information */
        printf(">Connected to
    }
```

```
    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);
    }
    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:
        SQLDisconnect(*hdbc);
        SQLFreeHandle(SQL_HANDLE_DBC, *hdbc);
        return (SQL_ERROR);
    } else {                    /* Print connection information */
        printf(">Connected to
    }
    return (SQL_SUCCESS);
}
/* ... */
```

*Figure 12. An application that connects to a data source*

**Related reference**

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

SQLDataSources() - Get a list of data sources
`SQLDataSources()` returns a list of available target databases, one at a time. Before you make a connection, you can call `SQLDataSources()` to determine which databases are available.

SQLDisconnect() - Disconnect from a data source
`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()`.

SQLDriverConnect() - Use a connection string to connect to a data source
`SQLDriverConnect()` is an alternative to `SQLConnect()`. Both functions establish a connection to the target database, but `SQLDriverConnect()` supports additional connection parameters.

SQLGetConnectOption() - Return current setting of a connect option
`SQLGetConnectOption()` is a deprecated function and is replaced by `SQLGetConnectAttr()`.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetConnectOption() - Set connection option
This function is deprecated and is replaced by SQLSetConnectAttr(). You cannot use `SQLSetConnectOption()` for 64-bit applications.

# SQLDataSources() - Get a list of data sources

`SQLDataSources()` returns a list of available target databases, one at a time. Before you make a connection, you can call `SQLDataSources()` to determine which databases are available.

### ODBC specifications for SQLDataSources()

*Table 69. SQLDataSources() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### 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

The following table lists the data type, use, and description for each argument in this function.

*Table 70. 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:<br>• SQL_FETCH_FIRST<br>• 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. |

*Table 70. SQLDataSources() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| 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 database management system.<br><br>**IBM specific:** IBM relational database management systems always return a blank description that is padded to 30 bytes. |
| SQLSMALLINT | *cbDescriptionMax* | input | Specifies the maximum length, in bytes, of the *szDescription* buffer.<br><br>**IBM specific:** Db2 for z/OS ODBC always returns NULL. |
| 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.<br><br>**IBM specific:** Db2 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

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 71. SQLDataSources() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | Data truncated. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• 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.<br>(SQLDataSources() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The specified value for either the *cbDSNMax*argument or the *cbDescriptionMax* argument is less than 0. |
| **HY**103 | Direction option out of range. | The *fDirection* argument is not set to SQL_FETCH_FIRST or SQL_FETCH_NEXT. |

**Example**

The following example 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 13. An application that lists available data sources*

**Related reference**

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLDescribeCol() - Describe column attributes

SQLDescribeCol() returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

### ODBC specifications for SQLDescribeCol()

*Table 72. SQLDescribeCol() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

For 31-bit applications, use the following 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);
```

For 64-bit applications, use the following syntax:

```
SQLRETURN   SQLDescribeCol   (SQLHSTMT          hstmt,
                              SQLUSMALLINT      icol,
                              SQLCHAR     FAR   *szColName,
                              SQLSMALLINT       cbColNameMax,
                              SQLSMALLINT FAR   *pcbColName,
                              SQLSMALLINT FAR   *pfSqlType,
                              SQLULEN     FAR   *pcbColDef,
                              SQLSMALLINT FAR   *pibScale,
                              SQLSMALLINT FAR   *pfNullable);
```

**Function arguments**

The following table lists the data type, use, and description for each argument in this function.

*Table 73. 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*. |
| 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. |
| SQLUINTEGER *(31-bit) or SQLULEN * (64-bit) [1] | *pcbColDef* | output | Returns the precision of the column as defined in the database. |
| SQLSMALLINT * | *pibScale* | output | Scale of column as defined in the database (applies only to SQL_DECIMAL, SQL_NUMERIC, and SQL_TYPE_TIMESTAMP. |
| SQLSMALLINT * | *pfNullable* | output | Indicates whether null values are allowed for the column with the following values:<br>• SQL_NO_NULLS<br>• SQL_NULLABLE |

**Notes:**

1. For 64-bit applications, the data type SQLUINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLULEN is recommended.

### Usage

If you need only one attribute of the descriptor information (column name, type, precision, scale, nullability), or if you need an attribute that `SQLDescribeCol()` does not return, use `SQLColAttribute()` in place of `SQLDescribeCol()`.

Usually, you call this function (or the `SQLColAttribute()` function) before you bind a column to an application variable.

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

### 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 the following table.

*Table 74. `SQLDescribeCol()` SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | 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) |
| **07**005 | 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.) |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |

*Table 74. SQLDescribeCol() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• The function is called prior to SQLPrepare() or SQLExecDirect() on the statement handle.<br>• 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The length that is specified in the *cbColNameMax* argument is less than 1. |
| **HY**C00 | Driver not capable. | Db2 ODBC does not recognize the SQL data type of column that the *icol* argument specifies. |
| **HY**002 | 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. |

**Example**

The following example 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);
    /* 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,
        } 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;
    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("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 size of buffer\n",
                        outlen[i], collen[i]);
            }
            if (outlen[i] == SQL_NULL_DATA)
                printf("i], collen[i], "NULL");
            else
                printf("i], collen[i], data[i]);
        }                       /* for all columns in this row  */
        printf("\n /* print any truncation messages    */
    }                           /* while rows to fetch */
    /* Free data buffers                                              */
    for (i = 0; i < nresultcols; i++) {
        free(data[i]);
    }
```

```
}                                   /* end display_results */
/* ... */
```

*Figure 14. An application that retrieves column descriptor information*

**Related reference**

C and SQL data types
Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

Precision of SQL data types
The precision of a numeric column or parameter refers to the maximum number of digits that are used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter.

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.

SQLBindCol() - Bind a column to an application variable
`SQLBindCol()` binds a column to an application variable. You can call `SQLBindCol()` once for each column in a result set from which you want to retrieve data or LOB locators.

SQLColAttribute() - Get column attributes
`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.

SQLExecDirect() - Execute a statement directly
`SQLExecDirect()` prepares and executes an SQL statement in one step.

SQLNumResultCols() - Get number of result columns
`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 you call `SQLNumResultCols()`. After you call `SQLNumResultCols()`, you can call `SQLColAttribute()` or one of the bind column functions.

SQLPrepare() - Prepare a statement
`SQLPrepare()` associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLDescribeParam() - Describe parameter marker

`SQLDescribeParam()` retrieves the description of a parameter marker that is associated with a prepared statement. This function is supported only for Db2 for z/OS data sources. Before you call this function, you must call either `SQLPrepare()` or `SQLExecDirect()`.

**ODBC specifications for SQLDescribeParam()**

*Table 75. SQLDescribeParam() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

For 31-bit applications, use the following syntax:

```
SQLRETURN  SQLDescribeParam  (SQLHSTMT          hstmt,
                              SQLUSMALLINT      ipar,
                              SQLSMALLINT FAR     *pfSqlType,
                              SQLUINTEGER FAR     *pcbColDef,
                              SQLSMALLINT FAR     *pibScale,
                              SQLSMALLINT FAR     *pfNullable);
```

For 64-bit applications, use the following syntax:

```
SQLRETURN  SQLDescribeParam  (SQLHSTMT          hstmt,
                              SQLUSMALLINT      ipar,
                              SQLSMALLINT FAR     *pfSqlType,
                              SQLULEN     FAR     *pcbColDef,
                              SQLSMALLINT FAR     *pibScale,
                              SQLSMALLINT FAR     *pfNullable);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 76. 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 * (31-bit) or SQLULEN *(64-bit) [1] | *pcbColDef* | output | Returns the precision of the parameter marker. |
| SQLSMALLINT * | *pibScale* | output | Returns the scale of the parameter marker. |
| SQLSMALLINT * | *pfNullable* | output | Indicates whether the parameter allows null values. This argument returns one of the following values:<br>• SQL_NO_NULLS: The parameter does not allow null values; this is the default.<br>• SQL_NULLABLE: The parameter allows null values.<br>• SQL_NULLABLE_UNKNOWN: The driver cannot determine whether the parameter allows null values. |

**Notes:**

1. For 64-bit applications, the data type SQLUINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLULEN is recommended.

### Usage

For distinct types, `SQLDescribeParam()` returns both base data types for the input parameter.

`SQLDescribeParam()` does not return an indication of whether a parameter in an SQL statement is for input, input/output, or output. Except in calls to stored procedures, all parameters in SQL statements are input parameters. To determine the type of each parameter in a call to a stored procedure, call `SQLProcedureColumns()`.

### Return codes

After you call SQLDescribeParam(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 77. SQLDescribeParam() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | 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. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**093 | 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. |
| **HY**C00 | Driver not capable. | The data source is not Db2 for z/OS or Db2 for Linux, UNIX, and Windows. |

### Related reference

C and SQL data types
Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

Precision of SQL data types
The precision of a numeric column or parameter refers to the maximum number of digits that are used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter.

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.

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator
SQLBindParameter() binds parameter markers to application variables and extends the capability of the SQLSetParam() function.

SQLCancel() - Cancel statement

SQLCancel() terminates an SQLExecDirect() or SQLExecute() sequence prematurely.

SQLExecDirect() - Execute a statement directly
SQLExecDirect() prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
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.

SQLPrepare() - Prepare a statement
SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLDisconnect() - Disconnect from a data source

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 callSQLFreeHandle().

### ODBC specifications for SQLDisconnect()

*Table 78. SQLDisconnect() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLDisconnect    (SQLHDBC          hdbc);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 79. 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

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 80. SQLDisconnect() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**002 | Disconnect error. | An error occurs during the disconnect. However, the disconnect succeeds. SQLDisconnect returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **08**003 | Connection is closed. | The specified connection in the *hdbc* argument is not open. |
| **25**000 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.<br>This error does not apply to stored procedures that are written in Db2 ODBC. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |

**Example**

Refer to SQLDriverConnect() for a related example.

**Related reference**

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

SQLConnect() - Connect to a data source
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 can call SQLConnect(). Subsequently, you must call SQLConnect() before you allocate a statement handle.

SQLDriverConnect() - Use a connection string to connect to a data source
SQLDriverConnect() is an alternative to SQLConnect(). Both functions establish a connection to the target database, but SQLDriverConnect() supports additional connection parameters.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLTransact() - Transaction management
`SQLTransact()` is a deprecated function and is replaced by `SQLEndTran()`.

# `SQLDriverConnect()` - Use a connection string to connect to a data source

`SQLDriverConnect()` is an alternative to `SQLConnect()`. Both functions establish a connection to the target database, but `SQLDriverConnect()` supports additional connection parameters.

### ODBC specifications for `SQLDriverConnect()`

*Table 81. `SQLDriverConnect()` specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### Syntax

```
SQLRETURN SQLDriverConnect   (SQLHDBC           hdbc,
                              SQLHWND           hwnd,
                              SQLCHAR      FAR  *szConnStrIn,
                              SQLSMALLINT       cbConnStrIn,
                              SQLCHAR      FAR  *szConnStrOut,
                              SQLSMALLINT       cbConnStrOutMax,
                              SQLSMALLINT  FAR  *pcbConnStrOut,
                              SQLUSMALLINT      fDriverCompletion);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 82. `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 168 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.<br><br>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 82. SQLDriverConnect() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLSMALLINT* | *pcbConnStrOut* | output | Points to a buffer that contains the total number of available bytes for the complete connection string.<br><br>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.<br><br>**IBM specific:** Db2 for z/OS supports only the value of SQL_DRIVER_NOPROMPT for this argument. The following values are **not** supported:<br><br>• SQL_DRIVER_PROMPT<br>• SQL_DRIVER_COMPLETE<br>• SQL_DRIVER_COMPLETE_REQUIRED |

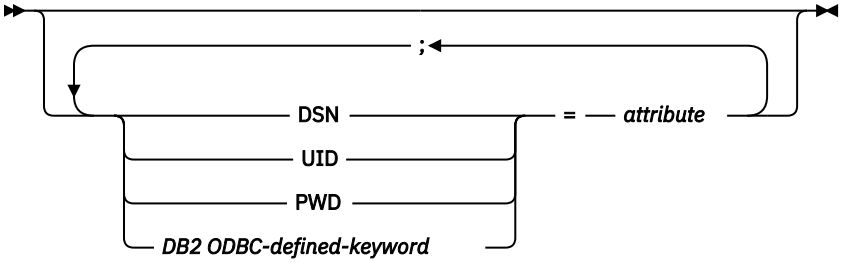
**Usage**

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.

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 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 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 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.

Any one of the initialization keywords 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

### Diagnostics

This function generates similar diagnostics as the function `SQLConnect()`. The following table shows the additional SQLSTATEs that `SQLDriverConnect()` returns.

*Table 83. `SQLDriverConnect()` SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | 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 83. SQLDriverConnect() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**S00 | 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:<br>• The unrecognized keyword is ignored.<br>• The invalid attribute value is ignored and the default value is used instead.<br>(SQLDriverConnect() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **01**S02 | Option value changed. | SQL_CONNECTTYPE changes to SQL_CONCURRENT_TRANS while MULTICONTEXT=1 is in use. |
| **HY**090 | Invalid string or buffer length. | This SQLSTATE is returned for one or more of the following reasons:<br>• The specified value for the *cbConnStrIn* argument is less than 0 and not equal to SQL_NTS.<br>• The specified value for the *cbConnStrOutMax* argument is less than 0. |
| **HY**110 | 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**

The following example 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;
  /*************************************************************/
  /* 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 =
     goto dberror;
   }
  /*************************************************************/
  /* Enumerate keywords and values returned from SQLDriverConnect  */
  /*************************************************************/
   (void) printf ("**** ConnStrOut =
   for (i = 1, token = strtok (ConnStrOut, ";");
        (token != NULL);
        token = strtok (NULL, ";"), i++)
     (void) printf ("**** Keyword #
  /*************************************************************/
  /* 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;
/*************************************************************/
  /* 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 15. An application that passes keyword values as it connects*

**Related reference**

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

SQLConnect() - Connect to a data source
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 can call SQLConnect(). Subsequently, you must call SQLConnect() before you allocate a statement handle.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

# SQLEndTran() - End transaction of a connection

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.

### ODBC specifications for SQLEndTran()

*Table 84. SQLEndTran() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

```
SQLRETURN  SQLEndTran (SQLSMALLINT       HandleType,
                       SQLHANDLE         Handle,
                       SQLSMALLINT       CompletionType);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 85. 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" on page 172 for more information. |
| SQLSMALLINT | *CompletionType* | input | Specifies whether to perform a commit or a rollback. Use one of the following values:<br>• SQL_COMMIT<br>• 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.

### Return codes

After you call SQLGetDiagRec(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 86. SQLEndTran() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | An informational message was generated. (SQLEndTran() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **08**003 | Connection is closed. | The connection handle is not in a connected state. |
| **08**007 | 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. |
| **40**001 | Transaction rollback. | The transaction is rolled back due to a resource deadlock with another transaction. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the memory that is required to support the execution or completion of the function. |
| **HY**010 | 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. |
| **HY**012 | Invalid transaction code. | The specified value for the *CompletionType* argument was neither SQL_COMMIT nor SQL_ROLLBACK. |
| **HY**092 | 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 DSN8P3VP sample application or online in the DSN1110.SDSNSAMP data set

**Related concepts**

DSN8O3VP sample application
The DSN8O3VP sample program validates the installation of Db2 ODBC.

When to call SQLEndTran()
In manual-commit mode, SQLEndTran() must be called before SQLDisconnect() is called.

**Related reference**

SQLFreeHandle() - Free a handle
SQLFreeHandle() frees an environment handle, a connection handle, or a statement handle.

SQLFreeStmt() - Free (or reset) a statement handle
SQLFreeStmt() ends processing for a statement, to which a statement handle refers. You can use it to close a cursor or drop the statement handle to free the Db2 ODBC resources that are associated with the statement handle. Call SQLFreeStmt() after you execute an SQL statement and process the results.

SQLGetInfo() - Get general information

SQLGetInfo() returns general information about the database management systems to which the application is currently connected. For example, SQLGetInfo() indicates which data conversions are supported.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLError() - Retrieve error information

SQLError() is a deprecated function and is replaced by SQLGetDiagRec().

### ODBC Specifications for SQLError()

*Table 87. SQLError() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | Yes |

### 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

The following table lists the data type, use, and description for each argument in this function.

*Table 88. 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 88. 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 database management system. If the error is generated by Db2 ODBC and not the database management system, 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:<br>`[DB2 for z/OS][CLI Driver]`<br>This preface indicates that Db2 ODBC detected the error and a connection to a database has not yet been made.<br>The error location, ERRLOC x:y:z, keyword value is embedded in the buffer also. This is an internal error code for diagnostics.<br>If the error is detected during a database connection, then the error message returned from the database management system is prefaced by:<br>`[DB2 for z/OS][CLI Driver][`*`database server-name`*`]`<br>*database management system-name* is the name that is returned by SQLGetInfo() with SQL_database management system_NAME information type.<br>For example,<br>`DB2`<br>`DB2/6000`<br>`Vendor`<br>`Vendor` indicates a non-IBM DRDA database management system.<br>If the error is generated by the database management system, 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. |

**Related reference**
SQLGetDiagRec() - Get multiple field settings of diagnostic record

SQLGetDiagRec() returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information. SQLGetDiagRec() also returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the error message text.

## SQLExecDirect() - Execute a statement directly

SQLExecDirect() prepares and executes an SQL statement in one step.

SQLExecDirect() uses the current values of the parameter marker variables, if any parameters exist in the statement. The statement can only be executed once.

### ODBC specifications for SQLExecDirect()

*Table 89. SQLExecDirect() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLExecDirect    (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szSqlStr,
                              SQLINTEGER         cbSqlStr);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 90. 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. Refer to SQLFreeStmt() for more information about how to free or reset a statement handle. |
| SQLCHAR * | *szSqlStr* | input | Specifies the string that contains the SQL statement. The connected database server must be able to prepare the statement. |
| 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 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().

To use SQLExecDirect(), the connected database server must be able to dynamically prepare statement.

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, 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.

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 SQLSetStmtAttr() 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.

You cannot specify the FOR *n* ROWS clause in a MERGE statement that you execute with SQLExecDirect(). Use SQLSetStmtAttr() with the SQL_ATTR_PARAMSET_SIZE statement attribute to specify the number of rows to merge.

**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

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

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 91. SQLExecDirect() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**504 | 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.) |
| **07**001 | 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. |
| **07**006 | Invalid conversion. | Transfer of data between Db2 ODBC and the application variables would result in incompatible data conversion. |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **21**S01 | 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. |
| **21**S02 | 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. |
| **22**001 | String data right truncation. | A character string that is assigned to a character type column exceeds the maximum length of the column. |
| **22**008 | Invalid datetime format or datetime field overflow. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.)<br>• Datetime field overflow occurred.<br>**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. |
| **22**012 | Division by zero is invalid. | The *szSqlStr* argument contains an SQL statement with an arithmetic expression that caused division by zero. |

*Table 91. SQLExecDirect() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **22**018 | Error in assignment. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• 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.<br>• 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. |
| **23**000 | Integrity constraint violation. | The execution of the SQL statement is not permitted because the execution would cause an integrity constraint violation in the database management system. |
| **24**000 | Invalid cursor state. | A cursor is open on the statement handle. |
| **24**504 | 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. |
| **34**000 | 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:<br>• A COMMIT<br>• A ROLLBACK<br>• An SQL statement that the connected database server could not prepare<br>• A statement containing a syntax error |
| **40**001 | Transaction rollback. | The transaction to which the SQL statement belongs is rolled back due to a deadlock or timeout. |
| **42***xxx*[1] | Syntax error or access rule violation | These SQLSTATEs indicate one of the following errors:<br>• For **42**5*xx*, the authorization ID does not have permission to execute the SQL statement that the *szSqlStr* argument contains.<br>• For **42***xxx*, a variety of syntax or access problems with the statement occur. |

*Table 91. SQLExecDirect() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **42**895 | 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:<br>• The LOB locator type that is specified on the bind parameter function call does not match the LOB data type of the parameter marker.<br>• 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. |
| **42**S01 | 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. |
| **42**S02 | Database object does not exist. | The *szSqlStr* argument contains an SQL statement that references a table name or view name that does not exist. |
| **42**S11 | Index already exists. | The *szSqlStr* argument contains a CREATE INDEX statement, and the specified index name already exists. |
| **42**S12 | Index not found. | The *szSqlStr* argument contains a DROP INDEX statement, and the specified index name does not exist. |
| **42**S21 | 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. |
| **42**S22 | Column not found. | The *szSqlStr* argument contains an SQL statement that references a column name that does not exist. |
| **44**000 | Integrity constraint violation. | When the *szSqlStr* argument contains an SQL statement with a parameter or literal, one of the following violations occur:<br>• The parameter value is NULL for a column that is defined as NOT NULL in the associated table column.<br>• A duplicate value is supplied for a column that is constrained to contain only unique values.<br>• An integrity constraint is violated. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**009 | Invalid use of a null pointer. | The *szSqlStr* argument specifies a null pointer. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |

*Table 91. SQLExecDirect() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**019 | Numeric value out of range. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• The *szSqlStr* argument contains an SQL statement with an arithmetic expression that causes division by zero. |
| **HY**090 | 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**'.

**Example**

Refer to SQLFetch() for a related example.

**Related concepts**

Differences between Db2 ODBC and embedded SQL
Even though key differences exist between Db2 ODBC and embedded SQL, Db2 ODBC can execute any SQL statements that can be prepared dynamically in embedded SQL.

Vendor escape clauses
Vendor escape clauses increase the portability of your application if your application accesses multiple data sources from different vendors. However, if your application accesses only Db2 data sources, you have no reason to use vendor escape clauses.

**Related reference**

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator
SQLBindParameter() binds parameter markers to application variables and extends the capability of the SQLSetParam() function.

SQLExecute() - Execute a statement
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.

SQLExtendedFetch() - Fetch an array of rows
SQLExtendedFetch() extends the function of SQLFetch() by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLFreeStmt() - Free (or reset) a statement handle
SQLFreeStmt() ends processing for a statement, to which a statement handle refers. You can use it to close a cursor or drop the statement handle to free the Db2 ODBC resources that are associated with the statement handle. Call SQLFreeStmt() after you execute an SQL statement and process the results.

SQLParamData() - Get next parameter for which a data value is needed
SQLParamData() is used in conjunction with SQLPutData() to send long data in pieces. You can also use this function to send fixed-length data.

SQLPrepare() - Prepare a statement

SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

SQLPutData() - Pass a data value for a parameter
SQLPutData() supplies 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. This result set is retrieved by the same functions that process a result set that is generated by a query.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetParam() - Bind a parameter marker to a buffer
SQLSetParam() is a deprecated function and is replaced with SQLBindParameter().

# SQLExecute() - Execute a statement

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.

### ODBC specifications for SQLExecute()

*Table 92. SQLExecute() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLExecute       (SQLHSTMT          hstmt);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 93. SQLExecute() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Specifies a statement handle. No open cursor can be associated with the statement handle; seeSQLFreeStmt() 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 SQLSetStmtAttr() 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.

**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, DELETE, or MERGE statement executed on the table, or on a view of the table.

**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.

The following table lists and describes the additional SQLSTATE that SQLExecute() can return.

*Table 94. SQLExecute() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**010 | Function sequence error. | SQLExecute() is called on a statement prior to SQLPrepare(). |

**Example**

Refer to SQLPrepare() for a related example.

**Related reference**

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator
SQLBindParameter() binds parameter markers to application variables and extends the capability of the SQLSetParam() function.

SQLExecDirect() - Execute a statement directly
SQLExecDirect() prepares and executes an SQL statement in one step.

SQLExtendedFetch() - Fetch an array of rows
SQLExtendedFetch() extends the function of SQLFetch() by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLFreeStmt() - Free (or reset) a statement handle
SQLFreeStmt() ends processing for a statement, to which a statement handle refers. You can use it to close a cursor or drop the statement handle to free the Db2 ODBC resources that are associated with the statement handle. Call SQLFreeStmt() after you execute an SQL statement and process the results.

SQLMoreResults() - Check for more result sets
SQLMoreResults() returns more information about a statement handle. The information can be associated with an array of input parameter values for a query, or a stored procedure that returns results sets.

SQLPrepare() - Prepare a statement
SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetParam() - Bind a parameter marker to a buffer
SQLSetParam() is a deprecated function and is replaced with SQLBindParameter().

SQLSetStmtAttr() - Set statement attributes

SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

## SQLExtendedFetch() - Fetch an array of rows

SQLExtendedFetch() extends the function of SQLFetch() by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

### ODBC specifications for SQLExtendedFetch()

*Table 95. SQLExtendedFetch() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | No | No |

### Syntax

For 31-bit applications, use the following syntax:

```
SQLRETURN SQLExtendedFetch  (SQLHSTMT           hstmt,
                             SQLUSMALLINT       fFetchType,
                             SQLINTEGER             irow,
                             SQLUINTEGER  FAR   *pcrow,
                             SQLUSMALLINT FAR   *rgfRowStatus);
```

For 64-bit applications, use the following syntax:

```
SQLRETURN SQLExtendedFetch  (SQLHSTMT           hstmt,
                             SQLUSMALLINT       fFetchType,
                             SQLLEN             irow,
                             SQLULEN      FAR   *pcrow,
                             SQLUSMALLINT FAR   *rgfRowStatus);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 96. 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 (31-bit) or SQLLEN (64-bit) [1] | *irow* | input | Reserved for future use. Use any integer for this argument. |

*Table 96. SQLExtendedFetch() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLUINTEGER *(31-bit) or SQLULEN * (64-bit)[2] | *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. |
| 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:<br>• SQL_ROW_SUCCESS<br>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.<br>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:<br>• SQL_ROW_DELETED<br>• SQL_ROW_UPDATED |

**Notes:**

1. For 64-bit applications, the data type SQLINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLLEN is recommended.
2. For 64-bit applications, the data type SQLUINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLULEN is recommended.

**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.

To fetch one row of data at a time, call SQLFetch() instead of SQLExtendedFetch().

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. The result set can be bound in a column-wise or row-wise fashion.

**Column-wise binding:** 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. To bind LOB column values to files, the application can call the SQLBindFileToCol() 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() points to is set to SQL_NULL_DATA.

**Row-wise binding:** 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().

**Error handling:** SQLExtendedFetch() returns errors in the same manner as SQLFetch() with the following exceptions:

- When a warning occurs that applies to a particular row in the rowset, SQLExtendedFetch() sets the corresponding entry in the row status array to SQL_ROW_SUCCESS, not SQL_ROW_SUCCESS_WITH_INFO.
- If errors occur in every row in the rowset, SQLExtendedFetch() returns SQL_SUCCESS_WITH_INFO, and not SQL_ERROR.
- In each group of status records that applies to an individual row, the first status record that is returned by SQLExtendedFetch() contains SQLSTATE 01S01 (error in row). If SQLExtendedFetch() cannot return additional SQLSTATEs, it returns only SQLSTATE 01S01.

**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.

#### 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

#### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 97. SQLExtendedFetch() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | Data truncated. | The data that is returned for one **or more** columns is truncated. (SQLExtendedFetch() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **01**S01 | Error in row. | An error occurs while fetching one or more rows. (SQLExtendedFetch() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **07**002 | Too many columns. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• 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. |
| **07**006 | Invalid conversion. | The data value can not be converted in a meaningful manner to the data type that the *fCType* argument in SQLBindCol() specifies. |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **22**002 | 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. |
| **22**008 | Invalid datetime format or datetime field overflow. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• The value of a date, time, or timestamp does not conform to the syntax for the data type that is specified.<br>• Datetime field overflow occurred.<br>**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. |
| **22**012 | Division by zero is invalid. | A value from an arithmetic expression is returned that results in division by zero. |
| **22**018 | Error in assignment. | This SQLSTATE is returned for one or more of the following reasons:<br>• A returned value is incompatible with the data type of the bound column.<br>• A returned LOB locator was incompatible with the data type of the bound column. |

*Table 97. SQLExtendedFetch() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **24**000 | Invalid cursor state. | The SQL statement that is executed on the statement handle is not a query. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• SQLExtendedFetch() is called on a statement handle after a SQLFetch() call, and before the SQLFreeStmt() (with the *fOption* argument set to SQL_CLOSE) call.<br>• The function is called prior to calling SQLPrepare() or SQLExecDirect() on the statement handle.<br>• 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**019 | Numeric value out of range. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• A value from an arithmetic expression is returned that results in division by zero. |
| **HY**106 | Fetch type out of range. | The value that the*fFetchType* argument specifies is not recognized. |
| **HY**C00 | Driver not capable. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• A call to SQLBindCol() is made for a column data type that Db2 ODBC does not support.<br>• The specified fetch type is recognized, but it is not supported. |

**Usage**

Although this function is deprecated in ODBC 3.0, this function is not deprecated in Db2 ODBC. However, ODBC applications should use SQLFetchScroll(), rather than SQLExtendedFetch().

**Example**

The following example 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;
/* ... */
    /*****************************************/
    /* 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("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);
     /*****************************************/
    /* 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("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 16. An application that performs an array fetch*

**Related concepts**

Retrieval of a result set into an array
An application can issue a query statement and fetch rows from the result set that the query generates.

**Related reference**

SQLBindCol() - Bind a column to an application variable
`SQLBindCol()` binds a column to an application variable. You can call `SQLBindCol()` once for each column in a result set from which you want to retrieve data or LOB locators.

SQLExecDirect() - Execute a statement directly
`SQLExecDirect()` prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
`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.

SQLFetch() - Fetch the next row
`SQLFetch()` advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLGetData() - Get data from a column
`SQLGetData()` retrieves data for a single column in the current row of the result set. You can also use `SQLGetData()` to retrieve large data values in pieces. After you call `SQLGetData()` for each column, call `SQLFetch()` or `SQLExtendedFetch()` for each row that you want to retrieve.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

# SQLFetch() - Fetch the next row

`SQLFetch()` advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

### ODBC specifications for SQLFetch()

*Table 98. SQLFetch() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLFetch         (SQLHSTMT          hstmt);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 99. SQLFetch() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Specifies the statement handle from which to fetch data. |

**Usage**

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().

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 194).

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.

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 100. SQLFetch() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | 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.) |
| **07**002 | Too many columns. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• 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. |

*Table 100. SQLFetch() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **07**006 | Invalid conversion. | The data value cannot be converted in a meaningful manner to the data type that the *fCType* argument in SQLBindCol() specifies. |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **22**002 | 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. |
| **22**008 | Invalid datetime format or datetime field overflow. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• The value of a date, time, or timestamp does not conform to the syntax for the specified data type.<br>• Datetime field overflow occurred.<br>**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. |
| **22**012 | Division by zero is invalid. | A value from an arithmetic expression is returned that results in division by zero. |
| **22**018 | Error in assignment. | This SQLSTATE is returned for one or more of the following reasons:<br>• A returned value is incompatible with the data type of binding.<br>• A returned LOB locator is incompatible with the data type of the bound column. |
| **24**000 | Invalid cursor state. | The previous SQL statement that is executed on the statement handle is not a query. |
| **54**028 | Maximum LOB locator assigned. | The maximum number of concurrent LOB locators has been reached. A new locator can not be assigned. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | 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:<br>• The specified column is less than 0 or greater than the number of result columns.<br>• The specified column is 0, but Db2 ODBC does not support ODBC bookmarks (*icol* = 0).<br>• SQLExtendedFetch() is called for this result set. |

*Table 100. SQLFetch() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• SQLFetch() is called for a statement handle after SQLExtendedFetch() and before SQLCloseCursor().<br>• The function is called prior to SQLPrepare() or SQLExecDirect().<br>• 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**019 | Numeric value out of range. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• A value from an arithmetic expression is returned that results in division by zero.<br>**Important:** The associated cursor is undefined if this error is detected by Db2 for z/OS. If the error is detected by Db2 for Linux, UNIX, and Windows or by other IBM relational database management systems, the cursor remains open and continues to advance on subsequent fetch calls. |
| **HY**C00 | Driver not capable. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• A call to SQLBindCol() was made for a column data type that is not supported by Db2 ODBC. |

**Example**

The following example 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 initialize 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("
    }
    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 17. An application that retrieves data from bound columns*

**Related concepts**

The ODBC row status array
The row status array returns the status of each row in the rowset.

**Related reference**

SQLExecDirect() - Execute a statement directly
`SQLExecDirect()` prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
`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.

SQLExtendedFetch() - Fetch an array of rows
`SQLExtendedFetch()` extends the function of `SQLFetch()` by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that `SQLExtendedFetch()` returns.

SQLGetData() - Get data from a column

SQLGetData() retrieves data for a single column in the current row of the result set. You can also use SQLGetData() to retrieve large data values in pieces. After you call SQLGetData() for each column, call SQLFetch() or SQLExtendedFetch() for each row that you want to retrieve.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLFetchScroll() - Fetch the next row

SQLFetchScroll() fetches the specified rowset of data from the result set of a query and returns data for all bound columns. Rowsets can be specified at an absolute position or a relative position.

### ODBC specifications for SQLFetchScroll()

*Table 101. SQLFetchScroll() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

For 31-bit applications, use the following syntax:

```
SQLRETURN SQLFetchScroll (
             SQLHSTMT StatementHandle,
             SQLUSMALLINT FetchOrientation,
             SQLINTEGER FetchOffset);
```

For 64-bit applications, use the following syntax:

```
SQLRETURN SQLFetchScroll (
             SQLHSTMT StatementHandle,
             SQLUSMALLINT FetchOrientation,
             SQLLEN FetchOffset);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 102. SQLFetchScroll() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Specifies the statement handle from which to fetch data. |
| SQLUSMALLINT | *FetchOrientation* | input | Type of fetch:<br>• SQL_FETCH_NEXT<br>• SQL_FETCH_PRIOR<br>• SQL_FETCH_FIRST<br>• SQL_FETCH_LAST<br>• SQL_FETCH_ABSOLUTE<br>• SQL_FETCH_RELATIVE<br>Db2 ODBC does not support SQL_FETCH_BOOKMARK.<br>See “Usage” on page 199 for more information. |

*Table 102. SQLFetchScroll() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLINTEGER (31-bit) or SQLLEN (64-bit) [1] | *FetchOffset* | input | Number of the row to fetch. The interpretation of this argument depends on the value of the *FetchOrientation* argument.<br>See "Usage" on page 199 for more information. |

**Notes:**

1. For 64-bit applications, the data type SQLINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLLEN is recommended.

**Usage**

SQLFetchScroll() returns a specified rowset from a result set. Rowsets can be specified by absolute or relative position. SQLFetchScroll() can be called only after a call that creates a result set and before the cursor over that result set is closed. If any columns are bound, SQLFetchScroll() returns the data in those columns. If the application specifies a pointer to a row status array or a buffer in which to return the number of rows that were fetched, SQLFetchScroll() returns this information . Calls to SQLFetchScroll() can be mixed with calls to SQLFetch(). An SQLFetch() call is equivalent to SQLFetchScroll() with a *FetchOrientation* value of SQL_FETCH_NEXT. SQLFetchScroll() calls cannot be mixed with SQLExtendedFetch() calls.

**How to position the cursor:** When a result set is created, the cursor is positioned before the start of the result set. SQLFetchScroll() positions the block cursor based on the values of the *FetchOrientation* and *FetchOffset* arguments. The rules for determining the start of the new rowset are shown in the next section.

The following table defines the *FetchOrientation* values.

*Table 103. Meanings of FetchOrientation values*

| FetchOrientation value | Meaning |
|---|---|
| SQL_FETCH_NEXT | Return the next rowset. This is equivalent to calling SQLFetch(). SQLFetchScroll() ignores the value of *FetchOffset*. |
| SQL_FETCH_PRIOR | Return the prior rowset. SQLFetchScroll() ignores the value of *FetchOffset*. |
| SQL_FETCH_RELATIVE | Return the rowset that is *FetchOffset* from the start of the current rowset. |
| SQL_FETCH_ABSOLUTE | Return the rowset that starts at row *FetchOffset*. |
| SQL_FETCH_FIRST | Return the first rowset in the result set. SQLFetchScroll() ignores the value of *FetchOffset*. |
| SQL_FETCH_LAST | Return the last complete rowset in the result set. SQLFetchScroll() ignores the value of *FetchOffset*. |

The SQL_ATTR_ROW_ARRAY_SIZE statement attribute specifies the number of rows in the rowset. If the rowset that is being fetched by SQLFetchScroll() goes beyond the end of the result set, SQLFetchScroll() returns a partial rowset. That is, if S is the starting row of the rowset, R is the rowset size, and L is the length of the result set, and S+R-1 is greater than L, only the first L-S+1 rows of the rowset are valid. The remaining rows are empty and have a status of SQL_ROW_NOROW.

After SQLFetchScroll() completes, the rowset cursor is positioned on the first row of the rowset.

**Cursor positioning rules:** The following information describes the rules for determining the start of the new rowset for each value of *FetchOrientation*. These rules use the following notation:

| Fetch orientation notation | Meaning |
|---|---|
| *BeforeStart* | The block cursor is positioned before the start of the result set. If the first row of the rowset is before the start of the result set, SQLFetchScroll() returns SQL_NO_DATA. |
| *AfterEnd* | The block cursor is positioned after the end of the result set. If the first row of the rowset is after the end of the result set, SQLFetchScroll() returns SQL_NO_DATA. |
| *CurrRowsetStart* | The number of the first row in the current rowset. |
| *LastResultRow* | The number of the last row in the result set. |
| *RowsetSize* | The rowset size. |
| *FetchOffset* | The value of the *FetchOffset* argument in the SQLFetchScroll() call. |

The following table describes the rules for determining the start of the new rowset when the *FetchOrientation* value is SQL_FETCH_NEXT.

*Table 104. Cursor position when SQLFetchScroll() parameter FetchOrientation is SQL_FETCH_NEXT*

| Condition | First row of the new rowset |
|---|---|
| *BeforeStart* | 1 |
| *CurrRowsetStart* + *RowsetSize* <= *LastResultRow* | *CurrRowsetStart* + *RowsetSize* |
| *CurrRowsetStart* + *RowsetSize* > *LastResultRow* | *AfterEnd* |
| *AfterEnd* | *AfterEnd* |

The following table describes the rules for determining the start of the new rowset when the *FetchOrientation* value is SQL_FETCH_PRIOR.

*Table 105. Cursor position when SQLFetchScroll() parameter FetchOrientation is SQL_FETCH_PRIOR*

| Condition | First row of the new rowset |
|---|---|
| *BeforeStart* | *BeforeStart* |
| *CurrRowsetStart* =1 | *BeforeStart* |
| 1<*CurrRowsetStart*<=*RowsetSize* | 1[“1” on page 200] |
| *CurrRowsetStart*>*RowsetSize* | *CurrRowsetStart*-*RowsetSize* |
| *AfterEnd* and *LastResultRow*<*RowsetSize* | 1[“1” on page 200] |
| *AfterEnd* and *LastResultRow*>=*RowsetSize* | *LastResultRow*-*RowsetSize*+1 |

**Note:**

1. SQLFetchScroll() returns SQLSTATE 01S06 (attempt to fetch before the result set returned the first rowset) and SQL_SUCCESS_WITH_INFO.

The following table describes the rules for determining the start of the new rowset when the *FetchOrientation* value is SQL_FETCH_RELATIVE.

*Table 106. Cursor position when SQLFetchScroll() parameter FetchOrientation is SQL_FETCH_RELATIVE*

| Condition | First row of the new rowset |
|---|---|
| *BeforeStart* AND *FetchOffset*>0 | "1" on page 201 |
| *AfterEnd* AND*FetchOffset*<0 | "1" on page 201 |
| *BeforeStart* AND*FetchOffset*<=0 | *BeforeStart* |
| *CurrRowsetStart*=1 AND*FetchOffset*< 0 | *BeforeStart* |
| *CurrRowsetStart*>1 AND*CurrRowsetStart* +*FetchOffset*< 1 AND ABS(*FetchOffset*)>*RowsetSize* | *BeforeStart* |
| *CurrRowsetStart*>1 AND*CurrRowsetStart* +*FetchOffset*< 1 AND ABS(*FetchOffset*)<=*RowsetSize* | 1"2" on page 201 |
| 1<=(*CurrRowsetStart* + *FetchOffset*)<=*LastResultRow* | *CurrRowsetStart*+*FetchOffset* |
| (*CurrRowsetStart* + *FetchOffset*)>*LastResultRow* | *AfterEnd* |
| *AfterEnd* AND *FetchOffset*>=0 | *AfterEnd* |
| *FetchOffset*=0 | Unchanged"3" on page 201 |

**Note:**

1. SQLFetchScroll() returns the same rowset that is returned when it is called with *FetchOrientation* set to SQL_FETCH_ABSOLUTE.
2. SQLFetchScroll() returns SQLSTATE 01S06 (attempt to fetch before the result set returned the first rowset) and SQL_SUCCESS_WITH_INFO.
3. This is a special command to fetch data again. If the cursor is a sensitive cursor, data is refetched from the base table. If the cursor is an insensitive cursor, the buffer remains unchanged. A cursor is insensitive for one of the following reasons:
   - The statement attribute for the associated statement is SQL_ATTR_CURSOR_SENSITIVITY or SQL_INSENSITIVE.
   - The query is read-only.

The following table describes the rules for determining the start of the new rowset when the *FetchOrientation* value is SQL_FETCH_ABSOLUTE.

*Table 107. Cursor position when SQLFetchScroll() parameter FetchOrientation is SQL_FETCH_ABSOLUTE*

| Condition | First row of the new rowset |
|---|---|
| *FetchOffset*<0 AND ABS(*FetchOffset*)<=*LastResultRow* | *LastResultRow*+*FetchOffset*+1 |
| *FetchOffset*<0 AND ABS(*FetchOffset*)>*LastResultRow* AND ABS(*FetchOffset*)>*RowSetSize* | *BeforeStart* |
| *FetchOffset*<0 AND ABS(*FetchOffset*)>*LastResultRow* AND ABS(*FetchOffset*)<=*RowSetSize* | 1"1" on page 202 |

*Table 107. Cursor position when SQLFetchScroll() parameter FetchOrientation is SQL_FETCH_ABSOLUTE (continued)*

| Condition | First row of the new rowset |
|---|---|
| *FetchOffset*=0 | *BeforeStart* |
| 1<=*FetchOffset*<=*LastResultRow* | *FetchOffset* |
| *FetchOffset*>*LastResultRow* | *AfterEnd* |

**Note:**

1. SQLFetchScroll() returns SQLSTATE 01S06 (attempt to fetch before the result set returned the first rowset) and SQL_SUCCESS_WITH_INFO.

The following table describes the rules for determining the start of the new rowset when the *FetchOrientation* value is SQL_FETCH_FIRST.

*Table 108. Cursor position when SQLFetchScroll() parameter FetchOrientation is SQL_FETCH_FIRST*

| Condition | First row of the new rowset |
|---|---|
| Any | 1 |

The following table describes the rules for determining the start of the new rowset when the *FetchOrientation* value is SQL_FETCH_LAST.

*Table 109. Cursor position when SQLFetchScroll() parameter FetchOrientation is SQL_FETCH_LAST*

| Condition | First row of the new rowset |
|---|---|
| *RowsetSize*<=*LastResultRow* | *LastResultRow*-*RowsetSize*+1 |
| *RowsetSize*>*LastResultRow* | 1 |

**Data in bound columns:** SQLFetchScroll() returns data in bound columns in the same way as SQLFetch(). If no columns are bound, SQLFetchScroll() does not return data, but moves the block cursor to the specified position. As with SQLFetch(), you can use SQLGetData() to retrieve the values for each column.

**Buffer addresses:** SQLFetchScroll() uses the same formula to determine the address of data and length and indicator buffers as SQLFetch().

**Error handling:** SQLFetchScroll() returns errors and warnings in the same manner as SQLFetch().

### Return codes

After you call SQLFetchScroll(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

### Diagnostics

The return code that is associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 110. SQLFetchScroll SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (Function returns SQL_SUCCESS_WITH_INFO.) |
| **01**004 | Data truncated. | String or binary data that was returned for a column resulted in the truncation of non-blank character data or non-NULL binary data. String values are right truncated. (Function returns SQL_SUCCESS_WITH_INFO.) |
| **01**S06 | Attempt to fetch before the result set returned the first rowset. | The requested rowset overlapped the start of the result set when the current position was beyond the first row, and either of the following conditions was true:<br>• *FetchOrientation* was SQL_PRIOR<br>• *FetchOrientation* was SQL_RELATIVE with a negative *FetchOffset* whose absolute value was less than or equal to the current SQL_ATTR_ROW_ARRAY_SIZE.<br>(Function returns SQL_SUCCESS_WITH_INFO.) |
| **01**S07 | Fractional truncation. | The data that was returned for a column was truncated. For numeric data types, the fractional part of the number was truncated. For time or timestamp data types, or interval data types with a time component, the fractional portion of the time was truncated. |
| **07**002 | Too many columns. | A column number that was specified in the binding of one or more columns was greater than the number of columns in the result set. |
| **07**006 | Invalid conversion. | A data value of a column in the result set could not be converted to the C data type that was specified by *TargetType* in `SQLBindCol()`. |
| **08**S01 | Communication link failure. | The communication link between Db2 ODBC and the data source to which it was connected failed before the function completed processing. |
| **22**001 | String data right truncation. | A variable-length bookmark that was returned for a row was truncated. |
| **22**002 | Invalid output or indicator buffer specified. | NULL data was fetched into a column whose *pcbValue*, which was set by `SQLBindCol()`, was a null pointer. |
| **22**003 | Numeric value out of range. | Data was not returned because returning the numeric value (as numeric or string) for one or more bound columns would have caused the whole (as opposed to fractional) part of the number to be truncated. |
| **22**007 | Invalid datetime format. | A character column in the result set was bound to a date, time, or timestamp C structure, and a value in the column was an invalid date, time, or timestamp. |
| **22**012 | Division by zero is invalid. | An arithmetic expression resulted in division by zero. |

*Table 110. SQLFetchScroll SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **22**018 | Invalid character value for cast specification. | One of the following conditions occurred:<br>• A character column in the result set was bound to a character C buffer, and the column contained a character for which there was no representation in the character set of the buffer.<br>• A character column in the result set was bound to an approximate numeric C buffer, and a value in the column could not be cast to a valid approximate numeric value.<br>• A character column in the result set was bound to an exact numeric C buffer and a value in the column could not be cast to a valid exact numeric value.<br>• A character column in the result set was bound to a datetime C buffer and a value in the column could not be cast to a valid datetime value. |
| **24**000 | Invalid cursor state. | The *StatementHandle* was in an executed state but no result set was associated with the *StatementHandle*. |
| **40**001 | Transaction rollback. | The transaction in which the fetch was executed was terminated to prevent deadlock. |
| **HY**000 | General error. | An error occurred for which there was no specific SQLSTATE. The error message that was returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause. |
| **HY**001 | Memory allocation failure. | Db2 ODBC was unable to allocate memory required to support execution or completion of the function. Process-level memory might have been exhausted for the application process. Consult the operating system configuration for information on process-level memory limitations. |
| **HY**008 | Operation was canceled. | Before the function completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. |
| **HY**010 | Function sequence error. | One of the following conditions occurred:<br>• The specified *StatementHandle* was not in an executed state. The function was called without a previous call of `SQLExecDirect()`, `SQLExecute()`, or a catalog function.<br>• `SQLExecute()` or `SQLExecDirect()` was called for the *StatementHandle* and returned SQL_NEED_DATA. `SQLFetchScroll()` was called before data was sent for all data-at-execution parameters or columns.<br>• `SQLFetchScroll()` was called for a *StatementHandle* after `SQLFetch()` was called, and before `SQLFreeStmt()` was called with the SQL_CLOSE option, or before `SQLMoreResults()` was called. The connection was to a down-level server.<br>• `SQLFetchScroll()` was called for a *StatementHandle* after `SQLExtendedFetch()` was called and before `SQLFreeStmt()` with SQL_CLOSE was called. |

*Table 110. SQLFetchScroll SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**106 | Fetch type out of range. | The value that was specified for the argument *FetchOrientation* was invalid. The value of the SQL_CURSOR_TYPE statement attribute was SQL_CURSOR_FORWARD_ONLY, and the value of argument *FetchOrientation* was not SQL_FETCH_NEXT. |
| **HY**C00 | Driver not capable. | The specified fetch type is not supported. The conversion that is specified by the combination of *TargetType* in `SQLBindCol()` and the SQL data type of the corresponding column is not supported. |

**Related concepts**

The ODBC row status array
The row status array returns the status of each row in the rowset.

Scrollable cursors in Db2 ODBC
Scrollable cursors let you move backward and forward in a query result set.

**Related reference**

SQLBindCol() - Bind a column to an application variable
`SQLBindCol()` binds a column to an application variable. You can call `SQLBindCol()` once for each column in a result set from which you want to retrieve data or LOB locators.

SQLCancel() - Cancel statement
`SQLCancel()` terminates an `SQLExecDirect()` or `SQLExecute()` sequence prematurely.

SQLDescribeCol() - Describe column attributes
`SQLDescribeCol()` returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either `SQLPrepare()` or `SQLExecDirect()`.

SQLExecDirect() - Execute a statement directly
`SQLExecDirect()` prepares and executes an SQL statement in one step.

SQLFetch() - Fetch the next row
`SQLFetch()` advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLNumResultCols() - Get number of result columns
`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 you call `SQLNumResultCols()`. After you call `SQLNumResultCols()`, you can call `SQLColAttribute()` or one of the bind column functions.

SQLSetPos - Set the cursor position in a rowset
`SQLSetPos()` sets the cursor position in a rowset. Once the cursor is set, the application can refresh, update, and delete data in the rows.

SQLSetStmtAttr() - Set statement attributes

SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

## SQLForeignKeys() - Get a list of foreign key columns

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 that is generated by a query.

### ODBC specifications for SQLForeignKeys()

*Table 111. SQLForeignKeys() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### 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

The following table lists the data type, use, and description for each argument in this function.

*Table 112. 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. |

*Table 112. SQLForeignKeys() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| 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. |
| 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.

The following table lists each column in the result set that SQLForeignKeys() currently returns.

*Table 113. 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. |

*Table 113. Columns returned by SQLForeignKeys() (continued)*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 9 | KEY_SEQ | SMALLINT NOT NULL | Contains the ordinal position of the column in the key. The first position is 1. |
| 10 | UPDATE_RULE | SMALLINT | Identifies the action that is applied to the foreign key when the SQL operation is UPDATE.<br>IBM Db2 database management systems always return one of the following values:<br>• SQL_RESTRICT<br>• SQL_NO_ACTION<br>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.<br>You might encounter the following UPDATE_RULE values when connected to non-IBM relational database management systems:<br>• SQL_CASCADE<br>• SQL_SET_NULL |
| 11 | DELETE_RULE | SMALLINT | Identifies the action that is applied to the foreign key when the SQL operation is DELETE.<br>The following values indicate the action that is applied:<br>• SQL_CASCADE: when a primary key value is deleted, that value in related foreign keys is also deleted.<br>• SQL_NO_ACTION: the delete is rejected if it removes values from a primary key that a foreign key references.<br>• SQL_RESTRICT: the delete is rejected if it removes values from a primary key that a foreign key references.<br>• SQL_SET_DEFAULT: when a primary key value is deleted, that value is replaced with a default value in related foreign keys.<br>• 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. |

*Table 113. Columns returned by SQLForeignKeys() (continued)*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 14 | DEFERRABILITY | SMALLINT | Db2 ODBC always returns a value of NULL.<br>Other database management systems support the following values:<br>• SQL_INITIALLY_DEFERRED<br>• SQL_INITIALLY_IMMEDIATE<br>• 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 for z/OS server should expect the result set columns listed in Table 113 on page 207.

For consistency with ANSI/ISO SQL standard of 1992 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 database management system:

- 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 114. SQLForeignKeys() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is open on the statement handle. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**009 | Invalid use of a null pointer. | The arguments *szPkTableName* and *szFkTableName* are both null pointers. |
| **HY**010 | 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.) |
| **HY**090 | Invalid string or buffer length. | This SQLSTATE is returned for one or more of the following reasons:<br>• The value of one of the name length arguments is less than 0 and not equal SQL_NTS.<br>• The length of the table or owner name is greater than the maximum length that is supported by the server. |
| **HY**C00 | Driver not capable. | Db2 ODBC does not support "catalog" as a qualifier for table name. |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |

### Example

The following example 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;
/*****************************************************************/
/* 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;
}
/*****************************************************************/
/* 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);
 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;
  /************************************************************/
  /* 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 18. An application that retrieves foreign key information about a table*

**Related reference**

SQLGetInfo() - Get general information
`SQLGetInfo()` returns general information about the database management systems to which the application is currently connected. For example, `SQLGetInfo()` indicates which data conversions are supported.

SQLPrimaryKeys() - Get primary key columns of a table
`SQLPrimaryKeys()` returns a list of column names that comprise the primary key for a table. The information is returned in an SQL result set. This result set can be retrieved by using the same functions that process a result set that is generated by a query.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLStatistics() - Get index and statistics information for a base table
`SQLStatistics()` retrieves index information for a specific table. It also returns the cardinality and the number of pages that are associated with the table and the indexes on the table. The information is returned in a result set. You can retrieve the result set with the same functions that process a result set that is generated by a query.

# SQLFreeConnect() - Free a connection handle

`SQLFreeConnect()` is a deprecated function and is replaced by `SQLFreeHandle()`.

**ODBC specifications for SQLFreeConnect()**

*Table 115. SQLFreeConnect() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | Yes |

### Syntax

```
SQLRETURN   SQLFreeConnect   (SQLHDBC          hdbc);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 116. SQLFreeConnect() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHDBC | *hdbc* | input | Connection handle |

### Related reference

SQLFreeHandle() - Free a handle
SQLFreeHandle() frees an environment handle, a connection handle, or a statement handle.

## SQLFreeEnv() - Free an environment handle

SQLFreeEnv() is a deprecated function and is replaced by SQLFreeHandle().

### ODBC specifications for SQLFreeEnv()

*Table 117. SQLFreeEnv() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | Yes |

### Syntax

```
SQLRETURN   SQLFreeEnv       (SQLHENV          henv);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 118. SQLFreeEnv arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHENV | *henv* | input | Environment handle |

### Related reference

SQLFreeHandle() - Free a handle

SQLFreeHandle() frees an environment handle, a connection handle, or a statement handle.

# SQLFreeHandle() - Free a handle

SQLFreeHandle() frees an environment handle, a connection handle, or a statement handle.

### ODBC specifications for SQLFreeHandle()

*Table 119. SQLFreeHandle() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

```
SQLRETURN  SQLFreeHandle     (SQLSMALLINT       HandleType,
                              SQLHANDLE         Handle);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 120. 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:<br>• SQL_HANDLE_ENV to free the environment handle<br>• SQL_HANDLE_DBC to free a connection handle<br>• 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.

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 121. SQLFreeHandle() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (SQLFreeHandle() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **08**003 | 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. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate memory that is required to support execution or completion of the function. |
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• 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. |
| **HY**013 | 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. |
| **HY**506 | Error closing a file. | An error is encountered when trying to close a temporary file. |

### Example

Refer to the DSN8O3VP sample application or online in the DSN1110.SDSNSAMP data set.

**Related concepts**

DSN8O3VP sample application
The DSN8O3VP sample program validates the installation of Db2 ODBC.

**Related reference**

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

SQLCancel() - Cancel statement
`SQLCancel()` terminates an `SQLExecDirect()` or `SQLExecute()` sequence prematurely.

SQLDisconnect() - Disconnect from a data source
`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()`.

SQLGetDiagRec() - Get multiple field settings of diagnostic record
`SQLGetDiagRec()` returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information. `SQLGetDiagRec()` also returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the error message text.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLFreeStmt() - Free (or reset) a statement handle

`SQLFreeStmt()` ends processing for a statement, to which a statement handle refers. You can use it to close a cursor or drop the statement handle to free the Db2 ODBC resources that are associated with the statement handle. Call `SQLFreeStmt()` after you execute an SQL statement and process the results.

### ODBC specifications for SQLFreeStmt()

*Table 122. SQLFreeStmt() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLFreeStmt      (SQLHSTMT          hstmt,
                              SQLUSMALLINT      fOption);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 123. SQLFreeStmt() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Specifies the statement handle that refers to the statement to be stopped. |

*Table 123. SQLFreeStmt() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLUSMALLINT | *fOption* | input | The following values specify the manner in which you free the statement handle:<br>• SQL_UNBIND<br>• SQL_RESET_PARAMS<br>• SQL_CLOSE<br>• SQL_DROP (Deprecated)<br>See "Usage" on page 218 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.

**SQL_DROP (Deprecated)**
In ODBC 3.0, SQLFreeHandle() with *HandleType* set to SQL_HANDLE_STMT replaces the SQL_DROP option of SQLFreeStmt().

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.

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.

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 124. SQLFreeStmt() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**092 | Option type out of range. | The specified value for the *fOption* argument is not one of the following values:<br>• SQL_CLOSE<br>• SQL_DROP<br>• SQL_UNBIND<br>• SQL_RESET_PARAMS |

### Example

Refer to SQLFetch() for a related example.

### Related reference

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

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator
SQLBindParameter() binds parameter markers to application variables and extends the capability of the SQLSetParam() function.

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

SQLExtendedFetch() - Fetch an array of rows
SQLExtendedFetch() extends the function of SQLFetch() by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

SQLFetch() - Fetch the next row

SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLFreeHandle() - Free a handle
SQLFreeHandle() frees an environment handle, a connection handle, or a statement handle.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetParam() - Bind a parameter marker to a buffer
SQLSetParam() is a deprecated function and is replaced with SQLBindParameter().

## SQLGetConnectAttr() - Get current attribute setting

SQLGetConnectAttr() returns the current setting of a connection attribute and also allows you to set these attributes.

### ODBC specifications for SQLGetConnectAttr()

*Table 125. SQLGetConnectAttr() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

```
SQLRETURN  SQLGetConnectAttr (SQLHDBC           ConnectionHandle,
                              SQLINTEGER        Attribute,
                              SQLPOINTER        ValuePtr,
                              SQLINTEGER        BufferLength,
                              SQLINTEGER        *StringLengthPtr);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 126. 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 SQLSetConnectAttr() 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.<br><br>**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. |

*Table 126. SQLGetConnectAttr() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| 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:<br>• For ODBC-defined attributes:<br>– If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*.<br>– If *ValuePtr* points to an integer, *BufferLength* is ignored.<br>• For driver-defined attributes (IBM extension):<br>– 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).<br>– If *ValuePtr* points to an integer, *BufferLength* is ignored. |
| 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:<br>• If *ValuePtr* is a null pointer, no length is returned.<br>• 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.<br>• 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().

**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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 127. SQLGetConnectAttr() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | An informational message. (SQLGetConnectAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **01**004 | 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.) |
| **08**003 | Connection is closed. | The *Attribute* argument specifies a value that requires an open connection, but the connection handle was not in a connected state. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate memory that is required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The value specified for the *BufferLength* argument is less than 0. |
| **HY**092 | Option type out of range. | The specified value for the *Attribute* argument is not valid for this version of Db2 ODBC. |
| **HY**C00 | 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. |

### 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 reference

SQLGetStmtAttr() - Get current setting of a statement attribute
SQLGetStmtAttr() returns the current setting of a statement attribute. To set these statement attributes, use SQLSetStmtAttr().

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetConnectAttr() - Set connection attributes
`SQLSetConnectAttr()` sets attributes that govern aspects of connections.

SQLSetStmtAttr() - Set statement attributes
`SQLSetStmtAttr()` sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call `SQLSetConnectAttr()`.

## SQLGetConnectOption() - Return current setting of a connect option

`SQLGetConnectOption()` is a deprecated function and is replaced by `SQLGetConnectAttr()`.

### ODBC specifications for SQLGetConnectOption()

*Table 128. SQLGetConnectOption() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | No |

### Syntax

```
SQLRETURN   SQLGetConnectOption (
                              SQLHDBC           hdbc,
                              SQLUSMALLINT      fOption,
                              SQLPOINTER        pvParam);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 129. SQLGetConnectOption() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| HDBC | *hdbc* | input | Connection handle. |
| SQLUSMALLINT | *fOption* | input | Attribute to set. SeeSQLSetConnectAttr() 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). |

### Related reference

SQLSetConnectAttr() - Set connection attributes
`SQLSetConnectAttr()` sets attributes that govern aspects of connections.

## SQLGetCursorName() - Get cursor name

`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.

### ODBC specifications for SQLGetCursorName()

*Table 130. SQLGetCursorName() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLGetCursorName (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCursor,
                              SQLSMALLINT       cbCursorMax,
                              SQLSMALLINT FAR   *pcbCursor);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 131. 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 225 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 132. SQLGetCursorName() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | 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.) |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**015 | 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(). |
| **HY**090 | Invalid string or buffer length. | The value specified for the *cbCursorMax*argument is less than 0. |
| **HY**092 | 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

The following example 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";
 (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;
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
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;
  }
  /*****************************************************************/
  /* 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 :
    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);
  }
  /*****************************************************************/
  /* 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 19. An application that extracts a cursor name*

**Related reference**

SQLExecDirect() - Execute a statement directly
`SQLExecDirect()` prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
`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.

SQLPrepare() - Prepare a statement
`SQLPrepare()` associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetCursorName() - Set cursor name
`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.

# SQLGetData() - Get data from a column

`SQLGetData()` retrieves data for a single column in the current row of the result set. You can also use `SQLGetData()` to retrieve large data values in pieces. After you call `SQLGetData()` for each column, call `SQLFetch()` or `SQLExtendedFetch()` for each row that you want to retrieve.

You must call `SQLFetch()` before `SQLGetData()`. 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.

### ODBC specifications for SQLGetData()

*Table 133. SQLGetData() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

For 31-bit applications, use the following syntax:

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

For 64-bit applications, use the following syntax:

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

**Function arguments**

The following table lists the data type, use, and description for each argument in this function.

*Table 134. 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. |
| SQLSMALLINT | *fCType* | input | Specifies the C data type of the column that *icol* indicates. You can specify the following types for the *fCType* argument:<br>• SQL_C_BIGINT<br>• SQL_C_BINARY<br>• SQL_C_BINARYXML<br>• SQL_C_BIT<br>• SQL_C_BLOB_LOCATOR<br>• SQL_C_CHAR<br>• SQL_C_CLOB_LOCATOR<br>• SQL_C_DBCHAR<br>• SQL_C_DBCLOB_LOCATOR<br>• SQL_C_DOUBLE<br>• SQL_C_FLOAT<br>• SQL_C_DECIMAL64<br>• SQL_C_DECIMAL128<br>• SQL_C_LONG<br>• SQL_C_SHORT<br>• SQL_C_TYPE_DATE<br>• SQL_C_TYPE_TIME<br>• SQL_C_TYPE_TIMESTAMP<br>• SQL_C_TYPE_TIMESTAMP_EXT<br>• SQL_C_TINYINT<br>• SQL_C_WCHAR<br>When you specify SQL_C_DEFAULT, data is converted to its default C data type. |
| SQLPOINTER | *rgbValue*[1] | output | Points to a buffer where the retrieved column data is stored. |
| SQLINTEGER (31-bit) or SQLLEN (64-bit)[2] | *cbValueMax* | input | Specifies the maximum size of the buffer to which the *rgbValue* argument points. |

*Table 134. SQLGetData() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLINTEGER * (31-bit) or SQLLEN * (64-bit)[2] | *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.<br><br>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.<br><br>The value is SQL_NO_TOTAL if truncation occurs and the ODBC driver cannot determine the number of bytes still available to return. SQLGetData() retrieves enough data to fill the buffer and returns SQL_NO_TOTAL as the length.<br><br>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. |

**Notes:**

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.
2. For 64-bit applications, the data type SQLINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLLEN is recommended.

**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, the CCSID that is specified for the CURRENTAPPENSCH keyword is an EBCDIC CCSID, 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 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, or the CCSID that is specified for the CURRENTAPPENSCH keyword is a Unicode CCSID (1200, 1208, 13488 or 17584).
  - 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, or the CCSID that is specified for the CURRENTAPPENSCH keyword is an ASCII CCSID.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLGetData() call.

**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 232 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" on page 231 and "3" on page 231 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.

If the preceding call to `SQLFetch()` failed, **do not** call `SQLGetData()`. In this case, `SQLGetData()` retrieves undefined data.

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 135. `SQLGetData()` SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | 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.) |
| **07**006 | Invalid conversion. | This SQLSTATE is returned for one or more of the following reasons:<br>• The data value cannot be converted to the C data type specified by the *fCType* argument.<br>• 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. |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **22**002 | 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. |

*Table 135. SQLGetData() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **22**008 | Invalid datetime format or datetime field overflow. | Datetime field overflow occurred.<br>**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. |
| **22**018 | Error in assignment. | A returned value is incompatible with the data type that the *fCType* argument denotes. |
| **24**000 | 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. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | 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:<br>• The specified column is less than 0 or greater than the number of result columns.<br>• The specified column is 0 (the *icol* argument is set to 0), but Db2 ODBC does not support ODBC bookmarks.<br>• SQLExtendedFetch() is called for this result set. |
| **HY**003 | Program type out of range. | The *fCType*argument specifies an invalid data type or SQL_C_DEFAULT. |
| **HY**009 | Invalid use of a null pointer. | This SQLSTATE is returned for one or more of the following reasons:<br>• The *rgbValue* argument specifies a null pointer.<br>• The *pcbValue* argument specifies a null pointer but the SQL data type of the column is graphic (DBCS).<br>• The *pcbValue* argument specifies a null pointer but the *fCType* argument specifies SQL_C_CHAR. |
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• The statement handle does not contain a cursor in a positioned state. SQLGetData() is called without first calling SQLFetch().<br>• 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**019 | 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. |

*Table 135. SQLGetData() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**090 | 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:<br>• SQL_C_CHAR<br>• SQL_C_BINARY<br>• SQL_C_DBCHAR<br>• SQL_C_DEFAULT (for the default types of SQL_C_CHAR, SQL_C_BINARY, or SQL_C_DBCHAR) |
| **HY**C00 | Driver not capable. | This SQLSTATE is returned for one or more of the following reasons:<br>• The SQL data type for the specified data type is recognized but Db2 ODBC does not support this data type.<br>• Db2 ODBC cannot perform the conversion between the SQL data type and application data type that is specified in the *fCType* argument.<br>• 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

The following example shows an application that uses SQLGetData() to retrieve data. You can compare this example with the one in SQLFetch() for a comparison in using bound columns.

```
 /*****************************************************************/
 /*        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");
 /*****************************************************************/
 /* 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;
  }
/*****************************************************************/
 /* 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);
  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
  /*****************************************************************/
  /* 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);
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
   (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 ("
   }
 }
/*****************************************************************/
/* 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;
/*****************************************************************/
/* 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;
  }
  /*****************************************************************/
  /* 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 20. An application that retrieves data using SQLGetData()*

**Related concepts**

Application encoding schemes and Db2 ODBC

Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The Db2 ODBC driver supports input and output character string arguments to ODBC APIs and input and output host variable data in each of these encoding schemes.

**Related reference**

C and SQL data types
Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

SQLExtendedFetch() - Fetch an array of rows
`SQLExtendedFetch()` extends the function of `SQLFetch()` by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that `SQLExtendedFetch()` returns.

SQLFetch() - Fetch the next row
`SQLFetch()` advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLGetDiagRec() - Get multiple field settings of diagnostic record

`SQLGetDiagRec()` returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information. `SQLGetDiagRec()` also returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the error message text.

### ODBC specifications for SQLGetDiagRec()

*Table 136. `SQLGetDiagRec()` specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

```
SQLRETURN  SQLGetDiagRec  (SQLSMALLINT      HandleType,
                           SQLHANDLE        Handle,
                           SQLSMALLINT      RecNumber,
                           SQLCHAR          *SQLState,
                           SQLINTEGER       *NativeErrorPtr,
                           SQLCHAR          *MessageText,
                           SQLSMALLINT      BufferLength,
                           SQLSMALLINT      *TextLengthPtr);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 137. `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:<br>• SQL_HANDLE_ENV for environment handles<br>• SQL_HANDLE_DBC for connection handles<br>• SQL_HANDLE_STMT for statement handles |

*Table 137. SQLGetDiagRec() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| 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. |
| SQLSMALLINT * | *TextLengthPtr* | output | Pointer to a buffer that contains the total number of bytes that are available in the buffer that the *MessageText* argument points to. The total number of available bytes does not include the number of bytes for nul-termination characters. 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" on page 241.

**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.

**Example**

Refer to the DSN8O3VP sample application or online in the data set DSN1110.SDSNSAMP

**Related concepts**

DSN8O3VP sample application
The DSN8O3VP sample program validates the installation of Db2 ODBC.

**Related reference**

SQLFreeHandle() - Free a handle
`SQLFreeHandle()` frees an environment handle, a connection handle, or a statement handle.

SQLFreeStmt() - Free (or reset) a statement handle
`SQLFreeStmt()` ends processing for a statement, to which a statement handle refers. You can use it to close a cursor or drop the statement handle to free the Db2 ODBC resources that are associated with the statement handle. Call `SQLFreeStmt()` after you execute an SQL statement and process the results.

SQLGetInfo() - Get general information

SQLGetInfo() returns general information about the database management systems to which the application is currently connected. For example, SQLGetInfo() indicates which data conversions are supported.

## SQLGetEnvAttr() - Return current setting of an environment attribute

SQLGetEnvAttr() returns the current setting for an environment attribute. You can also use the SQLSetEnvAttr() function to set these attributes.

### ODBC specifications for SQLGetEnvAttr()

*Table 138. SQLGetEnvAttr() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

```
SQLRETURN  SQLGetEnvAttr (SQLHENV           EnvironmentHandle,
                          SQLINTEGER        Attribute,
                          SQLPOINTER        ValuePtr,
                          SQLINTEGER        BufferLength,
                          SQLINTEGER        *StringLengthPtr);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 139. SQLGetEnvAttr() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHENV | *EnvironmentHandle* | input | Specifies the environment handle. |
| SQLINTEGER | *Attribute* | input | Specifies the attribute to retrieve. The list of environment attributes and their descriptions are described under the function SQLSetEnvAttr(). |
| 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:<br>• 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).<br>• If *ValuePtr* points to an integer, the *BufferLength* argument is ignored. |

*Table 139. SQLGetEnvAttr() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLINTEGER * | *StringLengthPtr* | output | Points to a buffer that contains the total number of bytes that are associated with the *ValuePtr* argument. This number does not include the number of bytes for nul-termination characters. 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.<br><br>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.

### Return codes

After you call SQLGetEnvAttr(), it returns one of the following values:

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 140. SQLGetEnvAttr() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate memory that is required to support execution or completion of the function. |
| **HY**092 | Option type out of range. | An invalid value for the *Attribute* argument is specified. |

### 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 reference

SQLAllocHandle() - Allocate a handle

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

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetEnvAttr() - Set environment attributes
SQLSetEnvAttr() sets attributes that affects all connections in an environment.

## SQLGetFunctions() - Get functions

SQLGetFunctions() indicates if a specific function is supported.

SQLGetFunctions() allows applications to adapt to different levels of support when they connect to different database servers. A connection to a database server must exist before SQLGetFunctions() is called.

### ODBC specifications for SQLGetFunctions()

*Table 141. SQLGetFunctions() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLGetFunctions  (SQLHDBC           hdbc,
                              SQLUSMALLINT      fFunction,
                              SQLUSMALLINT FAR  *pfExists);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 142. SQLGetFunctions() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHDBC | *hdbc* | input | Specifies a database connection handle. |
| SQLUSMALLINT | *fFunction* | input | Specifies which function is queried. Table 143 on page 245 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 143 on page 245 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 143. SQLGetFunctions() functions and values*

| *fFunction* | Value Db2 ODBC returns |
|---|---|
| SQL_API_SQLALLOCCONNECT | SQL_TRUE |
| SQL_API_SQLALLOCENV | SQL_TRUE |
| SQL_API_SQLALLOCHANDLE | SQL_TRUE |
| SQL_API_SQLALLOCSTMT | SQL_TRUE |
| SQL_API_SQLBINDCOL | SQL_TRUE |
| SQL_API_SQLBINDFILETOCOL | SQL_TRUE |
| SQL_API_SQLBINDFILETOPARAM | SQL_TRUE |
| SQL_API_SQLBINDPARAMETER | SQL_TRUE |
| SQL_API_SQLBROWSECONNECT | SQL_FALSE |
| SQL_API_SQLBULKOPERATIONS | SQL_TRUE |
| 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_SQLFETCHSCROLL | 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 |

*Table 143. SQLGetFunctions( ) functions and values (continued)*

| *fFunction* | Value Db2 ODBC returns |
|---|---|
| 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 |
| 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 |

*Table 143. SQLGetFunctions() functions and values (continued)*

| *fFunction* | Value Db2 ODBC returns |
|---|---|
| SQL_API_SQLSETPOS | SQL_TRUE |
| 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

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 144. SQLGetFunctions() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**009 | Invalid use of a null pointer. | The argument *pfExists* specifies a null pointer. |
| **HY**010 | Function sequence error. | SQLGetFunctions() is called before a database connection is established. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |

**Example**

The following example 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"    }
                                    } ;

  /*****************************************************************/
  /* 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;
   /*****************************************************************/
  /* 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 21. An application that checks the database server for API support*

**Related reference**

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLGetInfo() - Get general information

SQLGetInfo() returns general information about the database management systems to which the application is currently connected. For example, SQLGetInfo() indicates which data conversions are supported.

### ODBC specifications for SQLGetInfo()

*Table 145. SQLGetInfo() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN SQLGetInfo (SQLHDBC       ConnectionHandle,
                      SQLUSMALLINT  InfoType,
                      SQLPOINTER    InfoValuePtr,
                      SQLSMALLINT   BufferLength,
                      SQLSMALLINT   *FAR StringLengthPtr);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 146. 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 147 on page 250. |

*Table 146. SQLGetInfo() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| 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:<br>• 16-bit integer value<br>• 32-bit integer value<br>• 32-bit binary value<br>• 32-bit mask<br>• 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.<br>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 147 on page 250 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.

**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).

The following table specifies each value that you can specify for the *InfoType* argument and describes the information that each of these values will return.

*Table 147. 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. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| 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:<br>• SQL_AF_ALL<br>• SQL_AF_AVG<br>• SQL_AF_COUNT<br>• SQL_AF_DISTINCT<br>• SQL_AF_MAX<br>• SQL_AF_MIN<br>• 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:<br>• SQL_AD_ADD_CONSTRAINT_DEFERRABLE<br>• SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE<br>• SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED<br>• SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE<br>• SQL_AD_ADD_DOMAIN_CONSTRAINT<br>• SQL_AD_ADD_DOMAIN_DEFAULT<br>• SQL_AD_CONSTRAINT_NAME_DEFINITION<br>• SQL_AD_DROP_DOMAIN_CONSTRAINT<br>• SQL_AD_DROP_DOMAIN_DEFAULT |
| SQL_ALTER_TABLE | 32-bit mask | Indicates which clauses in ALTER TABLE are supported by the database management system.<br>• SQL_AT_ADD_COLUMN<br>• 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:<br>• SQL_BRC_PROCEDURES<br>• SQL_BRC_EXPLICIT |
| SQL_BATCH_SUPPORT | 32-bit mask | Indicates which level of batches are supported:<br>• SQL_BS_SELECT_EXPLICIT, supports explicit batches that can have result-set generating statements.<br>• SQL_BS_ROW_COUNT_EXPLICIT, supports explicit batches that can have row-count generating statements.<br>• SQL_BS_SELECT_PROC, supports explicit procedures that can have result-set generating statements.<br>• 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. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_CATALOG_LOCATION<br>(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<br>(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<br>(In previous versions of Db2 ODBC, this *InfoType* is SQL_QUALIFIER_TERM.) | string | The database vendor's terminology for a qualifier.<br>The name that the vendor uses for the high order part of a three part name.<br>Because Db2 ODBC does not support three part names, a zero-length string is returned.<br>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. |
| 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.<br>• SQL_CB_NULL - indicates the result is a null value (this is the case for IBM relational database management systems).<br>• SQL_CB_NON_NULL - indicates the result is a concatenation of non-null column values. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_CONVERT_BIGINT<br>SQL_CONVERT_BINARY<br>SQL_CONVERT_BIT<br>SQL_CONVERT_CHAR<br>SQL_CONVERT_DATE<br>SQL_CONVERT_DECIMAL<br>SQL_CONVERT_DOUBLE<br>SQL_CONVERT_FLOAT<br>SQL_CONVERT_INTEGER<br>SQL_CONVERT_INTERVAL_DAY_TIME<br>SQL_CONVERT_INTERVAL_YEAR_MONTH<br>SQL_CONVERT_LONGVARBINARY<br>SQL_CONVERT_LONGVARCHAR<br>SQL_CONVERT_NUMERIC<br>SQL_CONVERT_REAL<br>SQL_CONVERT_ROWID<br>SQL_CONVERT_SMALLINT<br>SQL_CONVERT_TIME<br>SQL_CONVERT_TIMESTAMP<br>SQL_CONVERT_TINYINT<br>SQL_CONVERT_VARBINARY<br>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.<br>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.<br>The following bit masks are used to determine which conversions are supported:<br>• SQL_CVT_BIGINT<br>• SQL_CVT_BINARY<br>• SQL_CVT_BIT<br>• SQL_CVT_CHAR<br>• SQL_CVT_DATE<br>• SQL_CVT_DECIMAL<br>• SQL_CVT_DOUBLE<br>• SQL_CVT_FLOAT<br>• SQL_CVT_INTEGER<br>• SQL_CVT_LONGVARBINARY<br>• SQL_CVT_LONGVARCHAR<br>• SQL_CVT_NUMERIC<br>• SQL_CVT_REAL<br>• SQL_CVT_ROWID<br>• SQL_CVT_SMALLINT<br>• SQL_CVT_TIME<br>• SQL_CVT_TIMESTAMP<br>• SQL_CVT_TINYINT<br>• SQL_CVT_VARBINARY<br>• SQL_CVT_VARCHAR |
| SQL_CONVERT_FUNCTIONS | 32-bit mask | Indicates the scalar conversion functions supported by the driver and associated data source.<br>• SQL_FN_CVT_CONVERT - used to determine which conversion functions are supported.<br>• SQL_FN_CVT_CAST - used to determine which cast functions are supported. |
| SQL_CORRELATION_NAME | 16-bit integer | Indicates the degree of correlation name support by the server:<br>• SQL_CN_ANY, supported and can be any valid user-defined name.<br>• SQL_CN_NONE, correlation name not supported.<br>• SQL_CN_DIFFERENT, correlation name supported but it must be different than the name of the table that it represents. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_CLOSE_BEHAVIOR | 32-bit integer | Indicates whether locks are released when the cursor is closed. The possible values are:<br>• SQL_CC_NO_RELEASE: locks are not released when the cursor on this statement handle is closed. This is the default.<br>• SQL_CC_RELEASE: locks are released when the cursor on this statement handle is closed.<br>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 database management system. Db2 ODBC always returns zero; the CREATE ASSERTION statement is not supported. ODBC also defines the following values that Db2 ODBC does not return:<br>• SQL_CA_CREATE_ASSERTION<br>• SQL_CA_CONSTRAINT_INITIALLY_DEFERRED<br>• SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE<br>• SQL_CA_CONSTRAINT_DEFERRABLE<br>• 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 database management system. 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:<br>• SQL_CCS_CREATE_CHARACTER_SET<br>• SQL_CCS_COLLATE_CLAUSE<br>• SQL_CCS_LIMITED_COLLATION |
| SQL_CREATE_COLLATION | 32-bit mask | Indicates which clauses in the CREATE COLLATION statement are supported by the database management system. Db2 ODBC always returns zero; the CREATE COLLATION statement is not supported. ODBC also defines the following values that Db2 ODBC does not return:<br>• SQL_CCOL_CREATE_COLLATION |
| SQL_CREATE_DOMAIN | 32-bit mask | Indicates which clauses in the CREATE DOMAIN statement are supported by the database management system. Db2 ODBC always returns zero; the CREATE DOMAIN statement is not supported. ODBC also defines the following values that Db2 ODBC does not return:<br>• SQL_CDO_CREATE_DOMAIN<br>• SQL_CDO_CONSTRAINT_NAME_DEFINITION<br>• SQL_CDO_DEFAULT<br>• SQL_CDO_CONSTRAINT<br>• SQL_CDO_COLLATION<br>• SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED<br>• SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE<br>• SQL_CDO_CONSTRAINT_DEFERRABLE<br>• SQL_CDO_CONSTRAINT_NON_DEFERRABLE |

*Table 147. 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 database management system:<br>• SQL_CS_CREATE_SCHEMA<br>• SQL_CS_AUTHORIZATION<br>• SQL_CS_DEFAULT_CHARACTER_SET |
| SQL_CREATE_TABLE | 32-bit mask | Indicates which clauses in the CREATE TABLE statement are supported by the database management system. The following bit masks are used to determine which clauses are supported:<br>• SQL_CT_CREATE_TABLE<br>• SQL_CT_TABLE_CONSTRAINT<br>• SQL_CT_CONSTRAINT_NAME_DEFINITION<br>The following bits specify the ability to create temporary tables:<br>• SQL_CT_COMMIT_PRESERVE, deleted rows are preserved on commit.<br>• SQL_CT_COMMIT_DELETE, deleted rows are deleted on commit.<br>• SQL_CT_GLOBAL_TEMPORARY, global temporary tables can be created.<br>• SQL_CT_LOCAL_TEMPORARY, local temporary tables can be created.<br>The following bits specify the ability to create column constraints:<br>• SQL_CT_COLUMN_CONSTRAINT, specifying column constraints is supported.<br>• SQL_CT_COLUMN_DEFAULT, specifying column defaults is supported.<br>• SQL_CT_COLUMN_COLLATION, specifying column collation is supported.<br>The following bits specify the supported constraint attributes if specifying column or table constraints is supported:<br>• SQL_CT_CONSTRAINT_INITIALLY_DEFERRED<br>• SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE<br>• SQL_CT_CONSTRAINT_DEFERRABLE<br>• SQL_CT_CONSTRAINT_NON_DEFERRABLE |
| SQL_CREATE_TRANSLATION | 32-bit mask | Indicates which clauses in the CREATE TRANSLATION statement are supported by the database management system. Db2 ODBC always returns zero; the CREATE TRANSLATION statement is not supported. ODBC also defines the following value that Db2 ODBC does not return:<br>• SQL_CTR_CREATE_TRANSLATION |
| SQL_CURSOR_COMMIT_BEHAVIOR | 16-bit integer | Indicates how a COMMIT operation affects cursors. A value of:<br>• SQL_CB_DELETE, destroys cursors and drops access plans for dynamic SQL statements.<br>• SQL_CB_CLOSE, destroys cursors, but retains access plans for dynamic SQL statements (including non-query statements)<br>• 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.<br>After COMMIT, a FETCH must be issued to reposition the cursor before actions such as positioned updates or deletes can be taken. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_CURSOR_ROLLBACK_BEHAVIOR | 16-bit integer | Indicates how a ROLLBACK operation affects cursors. A value of:<br>• SQL_CB_DELETE, destroys cursors and drops access plans for dynamic SQL statements.<br>• SQL_CB_CLOSE, destroys cursors, but retains access plans for dynamic SQL statements (including non-query statements)<br>• 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.<br>Db2 servers do not have the SQL_CB_PRESERVE property. |
| SQL_CURSOR_SENSITIVITY | 32-bit unsigned integer | Indicates support for cursor sensitivity:<br>• SQL_INSENSITIVE<br>All cursors on the statement handle show the result set without reflecting any changes made to the result set by any other cursor within the same transaction.<br>• SQL_UNSPECIFIED<br>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.<br>• SQL_SENSITIVE<br>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 database management systems. |
| SQL_database server_NAME | string | The name of the database management system product being accessed. |
| SQL_database server_VER | string | The version of the database management system 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:<br>• SQL_DI_CREATE_INDEX<br>• SQL_DI_DROP_INDEX |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_DEFAULT_TXN_ISOLATION | 32-bit mask | The default transaction isolation level supported.<br>One of the following masks are returned:<br>• SQL_TXN_READ_UNCOMMITTED = Changes are immediately perceived by all transactions (dirty read, non-repeatable read, and phantoms are possible).<br>This is equivalent to the IBM UR level.<br>• SQL_TXN_READ_COMMITTED = Row read by transaction 1 can be altered and committed by transaction 2 (non-repeatable read and phantoms are possible)<br>This is equivalent to the IBM CS level.<br>• 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)<br>This is equivalent to the IBM RS level.<br>• SQL_TXN_SERIALIZABLE = Data affected by pending transaction is not available to other transactions (repeatable read, phantoms are not possible)<br>This is equivalent to the IBM RR level.<br>• SQL_TXN_VERSIONING = Not applicable to IBM database management systems.<br>• SQL_TXN_NOCOMMIT = Any changes are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed.<br>This is a DB2 for i isolation level.<br>In IBM terminology,<br>• SQL_TXN_READ_UNCOMMITTED is uncommitted read;<br>• SQL_TXN_READ_COMMITTED is cursor stability;<br>• SQL_TXN_REPEATABLE_READ is read stability;<br>• 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*'<br>***mm***<br>The major version.<br>***vv***<br>The minor version.<br>***rrrr***<br>The release.<br>For example, '08.01.0000' means, " major version 3, minor version 1, release 0." |

*Table 147. 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 database management system. Db2 ODBC always returns zero; the DROP ASSERTION statement is not supported. ODBC also defines the following value that Db2 ODBC does not return:<br>• SQL_DA_DROP_ASSERTION |
| SQL_DROP_CHARACTER_SET | 32-bit mask | Indicates which clause in the DROP CHARACTER SET statement is supported by the database management system. 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.<br>• SQL_DCS_DROP_CHARACTER_SET |
| SQL_DROP_COLLATION | 32-bit mask | Indicates which clause in the DROP COLLATION statement is supported by the database management system. Db2 ODBC always returns zero; the DROP COLLATION statement is not supported. ODBC also defines the following value that Db2 ODBC does not return:<br>• SQL_DC_DROP_COLLATION |
| SQL_DROP_DOMAIN | 32-bit mask | Indicates which clauses in the DROP DOMAIN statement are supported by the database management system. Db2 ODBC always returns zero; the DROP DOMAIN statement is not supported. ODBC also defines the following values that Db2 ODBC does not return:<br>• SQL_DD_DROP_DOMAIN<br>• SQL_DD_CASCADE<br>• SQL_DD_RESTRICT |
| SQL_DROP_SCHEMA | 32-bit mask | Indicates which clauses in the DROP SCHEMA statement are supported by the database management system.<br>• SQL_DS_DROP_SCHEMA<br>• SQL_DS_CASCADE<br>• SQL_DS_RESTRICT |
| SQL_DROP_TABLE | 32-bit mask | Indicates which clauses in the DROP TABLE statement are supported by the database management system:<br>• SQL_DT_DROP_TABLE<br>• SQL_DT_CASCADE<br>• SQL_DT_RESTRICT |
| SQL_DROP_TRANSLATION | 32-bit mask | Indicates which clauses in the DROP TRANSLATION statement are supported by the database management system. Db2 ODBC always returns zero; the DROP TRANSLATION statement is not supported. ODBC also defines the following value that Db2 ODBC does not return:<br>• SQL_DTR_DROP_TRANSLATION |
| SQL_DROP_VIEW | 32-bit mask | Indicates which clauses in the DROP VIEW statement are supported by the database management system.<br>• SQL_DV_DROP_VIEW<br>• SQL_DV_CASCADE<br>• SQL_DV_RESTRICT |

*Table 147. 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).<br>• SQL_CA1_NEXT<br>• SQL_CA1_ABSOLUTE<br>• SQL_CA1_RELATIVE<br>• SQL_CA1_BOOKMARK<br>• SQL_CA1_LOCK_EXCLUSIVE<br>• SQL_CA1_LOCK_NO_CHANGE<br>• SQL_CA1_LOCK_NOLOCK<br>• SQL_CA1_POS_POSITION<br>• SQL_CA1_POS_UPDATE<br>• SQL_CA1_POS_DELETE<br>• SQL_CA1_POS_REFRESH<br>• SQL_CA1_POSITIONED_UPDATE<br>• SQL_CA1_POSITIONED_DELETE<br>• SQL_CA1_SELECT_FOR_UPDATE<br>• SQL_CA1_BULK_ADD<br>• SQL_CA1_BULK_UPDATE_BY_BOOKMARK<br>• SQL_CA1_BULK_DELETE_BY_BOOKMARK<br>• 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).<br>• SQL_CA2_READ_ONLY_CONCURRENCY<br>• SQL_CA2_LOCK_CONCURRENCY<br>• SQL_CA2_OPT_ROWVER_CONCURRENCY<br>• SQL_CA2_OPT_VALUES_CONCURRENCY<br>• SQL_CA2_SENSITIVITY_ADDITIONS<br>• SQL_CA2_SENSITIVITY_DELETIONS<br>• SQL_CA2_SENSITIVITY_UPDATES<br>• SQL_CA2_MAX_ROWS_SELECT<br>• SQL_CA2_MAX_ROWS_INSERT<br>• SQL_CA2_MAX_ROWS_DELETE<br>• SQL_CA2_MAX_ROWS_UPDATE<br>• SQL_CA2_MAX_ROWS_CATALOG<br>• SQL_CA2_MAX_ROWS_AFFECTS_ALL<br>• SQL_CA2_CRC_EXACT<br>• SQL_CA2_CRC_APPROXIMATE<br>• SQL_CA2_SIMULATE_NON_UNIQUE<br>• SQL_CA2_SIMULATE_TRY_UNIQUE<br>• 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. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_FETCH_DIRECTION | 32-bit mask | The supported fetch directions.<br>The following bit-masks are used in conjunction with the flag to determine which attribute values are supported.<br>• SQL_FD_FETCH_NEXT<br>• SQL_FD_FETCH_FIRST<br>• SQL_FD_FETCH_LAST<br>• SQL_FD_FETCH_PREV<br>• SQL_FD_FETCH_ABSOLUTE<br>• SQL_FD_FETCH_RELATIVE<br>• SQL_FD_FETCH_RESUME |
| SQL_FILE_USAGE | 16-bit integer | Reserved. Zero is returned. |
| SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1 | 32-bit mask | Indicates the attributes of a forward-only cursor that Db2 ODBC supports (subset 1 of 2).<br>• SQL_CA1_NEXT<br>• SQL_CA1_POSITIONED_UPDATE<br>• SQL_CA1_POSITIONED_DELETE<br>• SQL_CA1_SELECT_FOR_UPDATE<br>• SQL_CA1_LOCK_EXCLUSIVE<br>• SQL_CA1_LOCK_NO_CHANGE<br>• SQL_CA1_LOCK_NOLOCK<br>• SQL_CA1_POS_POSITION<br>• SQL_CA1_POS_UPDATE<br>• SQL_CA1_POS_DELETE<br>• SQL_CA1_POS_REFRESH<br>• SQL_CA1_BULK_ADD<br>• SQL_CA1_BULK_UPDATE_BY_BOOKMARK<br>• SQL_CA1_BULK_DELETE_BY_BOOKMARK<br>• 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).<br>• SQL_CA2_READ_ONLY_CONCURRENCY<br>• SQL_CA2_LOCK_CONCURRENCY<br>• SQL_CA2_MAX_ROWS_SELECT<br>• SQL_CA2_MAX_ROWS_CATALOG<br>• SQL_CA2_OPT_ROWVER_CONCURRENCY<br>• SQL_CA2_OPT_VALUES_CONCURRENCY<br>• SQL_CA2_SENSITIVITY_ADDITIONS<br>• SQL_CA2_SENSITIVITY_DELETIONS<br>• SQL_CA2_SENSITIVITY_UPDATES<br>• SQL_CA2_MAX_ROWS_INSERT<br>• SQL_CA2_MAX_ROWS_DELETE<br>• SQL_CA2_MAX_ROWS_UPDATE<br>• SQL_CA2_MAX_ROWS_AFFECTS_ALL<br>• SQL_CA2_CRC_EXACT<br>• SQL_CA2_CRC_APPROXIMATE<br>• SQL_CA2_SIMULATE_NON_UNIQUE<br>• SQL_CA2_SIMULATE_TRY_UNIQUE<br>• SQL_CA2_SIMULATE_UNIQUE |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| 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:<br>• SQL_GD_ANY_COLUMN<br>SQLGetData() can be called for unbound columns that precede the last bound column.<br>• SQL_GD_ANY_ORDER<br>SQLGetData() can be called for columns in any order.<br>ODBC also defines the following extensions, which are not returned by Db2 ODBC:<br>• SQL_GD_BLOCK<br>• SQL_GD_BOUND |
| SQL_GROUP_BY | 16-bit integer | Indicates the degree of support for the GROUP BY clause by the server:<br>• SQL_GB_NO_RELATION, the columns in the GROUP BY and in the SELECT list are not related<br>• SQL_GB_NOT_SUPPORTED, GROUP BY not supported<br>• SQL_GB_GROUP_BY_EQUALS_SELECT, GROUP BY must include all non-aggregated columns in the select list<br>• SQL_GB_GROUP_BY_CONTAINS_SELECT, the GROUP BY clause must contain all non-aggregated columns in the SELECT list |
| SQL_IDENTIFIER_CASE | 16-bit integer | Indicates case sensitivity of object names (such as table-name).<br>A value of:<br>• SQL_IC_UPPER = identifier names are stored in upper case in the system catalog.<br>• SQL_IC_LOWER = identifier names are stored in lower case in the system catalog.<br>• SQL_IC_SENSITIVE = identifier names are case sensitive, and are stored in mixed case in the system catalog.<br>• SQL_IC_MIXED = identifier names are not case sensitive, and are stored in mixed case in the system catalog.<br>**IBM specific:** Identifier names in IBM database management systems are not case sensitive. |
| SQL_IDENTIFIER_QUOTE_CHAR | string | Indicates the character used to surround a delimited identifier. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| 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:<br>• SQL_ISV_ASSERTIONS<br>• SQL_ISV_CHARACTER_SETS<br>• SQL_ISV_CHECK_CONSTRAINTS<br>• SQL_ISV_COLLATIONS<br>• SQL_ISV_COLUMN_DOMAIN_USAGE<br>• SQL_ISV_COLUMN_PRIVILEGES<br>• SQL_ISV_COLUMNS<br>• SQL_ISV_CONSTRAINT_COLUMN_USAGE<br>• SQL_ISV_CONSTRAINT_TABLE_USAGE<br>• SQL_ISV_DOMAIN_CONSTRAINTS<br>• SQL_ISV_DOMAINS<br>• SQL_ISV_KEY_COLUMN_USAGE<br>• SQL_ISV_REFERENTIAL_CONSTRAINTS<br>• SQL_ISV_SCHEMATA<br>• SQL_ISV_SQL_LANGUAGES<br>• SQL_ISV_TABLE_CONSTRAINTS<br>• SQL_ISV_TABLE_PRIVILEGES<br>• SQL_ISV_TABLES<br>• SQL_ISV_TRANSLATIONS<br>• SQL_ISV_USAGE_PRIVILEGES<br>• SQL_ISV_VIEW_COLUMN_USAGE<br>• SQL_ISV_VIEW_TABLE_USAGE<br>• SQL_ISV_VIEWS |
| SQL_INSERT_STATEMENT | 32-bit mask | Indicates support for INSERT statements:<br>• SQL_IS_INSERT_LITERALS<br>• SQL_IS_INSERT_SEARCHED<br>• 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 147. 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).<br>• SQL_CA1_NEXT<br>• SQL_CA1_ABSOLUTE<br>• SQL_CA1_RELATIVE<br>• SQL_CA1_BOOKMARK<br>• SQL_CA1_LOCK_EXCLUSIVE<br>• SQL_CA1_LOCK_NO_CHANGE<br>• SQL_CA1_LOCK_NOLOCK<br>• SQL_CA1_POS_POSITION<br>• SQL_CA1_POS_UPDATE<br>• SQL_CA1_POS_DELETE<br>• SQL_CA1_POS_REFRESH<br>• SQL_CA1_POSITIONED_UPDATE<br>• SQL_CA1_POSITIONED_DELETE<br>• SQL_CA1_SELECT_FOR_UPDATE<br>• SQL_CA1_BULK_ADD<br>• SQL_CA1_BULK_UPDATE_BY_BOOKMARK<br>• SQL_CA1_BULK_DELETE_BY_BOOKMARK<br>• 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).<br>• SQL_CA2_READ_ONLY_CONCURRENCY<br>• SQL_CA2_LOCK_CONCURRENCY<br>• SQL_CA2_OPT_ROWVER_CONCURRENCY<br>• SQL_CA2_OPT_VALUES_CONCURRENCY<br>• SQL_CA2_SENSITIVITY_ADDITIONS<br>• SQL_CA2_SENSITIVITY_DELETIONS<br>• SQL_CA2_SENSITIVITY_UPDATES<br>• SQL_CA2_MAX_ROWS_SELECT<br>• SQL_CA2_MAX_ROWS_INSERT<br>• SQL_CA2_MAX_ROWS_DELETE<br>• SQL_CA2_MAX_ROWS_UPDATE<br>• SQL_CA2_MAX_ROWS_CATALOG<br>• SQL_CA2_MAX_ROWS_AFFECTS_ALL<br>• SQL_CA2_CRC_EXACT<br>• SQL_CA2_CRC_APPROXIMATE<br>• SQL_CA2_SIMULATE_NON_UNIQUE<br>• SQL_CA2_SIMULATE_TRY_UNIQUE<br>• SQL_CA2_SIMULATE_UNIQUE |
| SQL_KEYWORDS | string | A string of all the keywords at the database management system 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. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| 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. |
| 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.<br>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.<br>Zero is returned, indicating that the limit is dependent on system resources.<br>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. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_MAX_TABLE_NAME_LEN | 16-bit integer | The maximum length of a table name (in bytes). |
| SQL_MAX_TABLES_IN_SELECT | 16-bit integer | Indicates the maximum number of table names allowed in a FROM clause in a <query specification>. |
| SQL_MAX_USER_NAME_LEN | 16-bit integer | Indicates the maximum size allowed for a <user identifier> (in bytes). |
| SQL_MULT_RESULT_SETS | string | The character string 'Y' indicates that the database supports multiple result sets, 'N' indicates that it does not. |
| SQL_MULTIPLE_ACTIVE_TXN | string | The character string 'Y' indicates that active transactions on multiple connections are allowed. 'N' indicates that only one connection at a time can have an active transaction. |
| 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:<br>• SQL_NNC_NON_NULL, columns can be defined as NOT NULL.<br>• 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:<br>• SQL_NC_HIGH, null values sort high<br>• 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.<br>The following bit masks are used to determine which numeric functions are supported:<br>• SQL_FN_NUM_ABS<br>• SQL_FN_NUM_ACOS<br>• SQL_FN_NUM_ASIN<br>• SQL_FN_NUM_ATAN<br>• SQL_FN_NUM_ATAN2<br>• SQL_FN_NUM_CEILING<br>• SQL_FN_NUM_COS<br>• SQL_FN_NUM_COT<br>• SQL_FN_NUM_DEGREES<br>• SQL_FN_NUM_EXP<br>• SQL_FN_NUM_FLOOR<br>• SQL_FN_NUM_LOG<br>• SQL_FN_NUM_LOG10<br>• SQL_FN_NUM_MOD<br>• SQL_FN_NUM_PI<br>• SQL_FN_NUM_POWER<br>• SQL_FN_NUM_RADIANS<br>• SQL_FN_NUM_RAND<br>• SQL_FN_NUM_ROUND<br>• SQL_FN_NUM_SIGN<br>• SQL_FN_NUM_SIN<br>• SQL_FN_NUM_SQRT<br>• SQL_FN_NUM_TAN<br>• SQL_FN_NUM_TRUNCATE |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_ODBC_API_CONFORMANCE | 16-bit integer | The level of ODBC conformance.<br>• SQL_OAC_NONE<br>• SQL_OAC_LEVEL1<br>• SQL_OAC_LEVEL2 |
| SQL_ODBC_SAG_CLI_CONFORMANCE | 16-bit integer | The compliance to the functions of the SQL Access Group (SAG) CLI specification.<br>A value of:<br>• SQL_OSCC_NOT_COMPLIANT - the driver is not SAG-compliant.<br>• SQL_OSCC_COMPLIANT - the driver is SAG-compliant. |
| SQL_ODBC_SQL_CONFORMANCE | 16-bit integer | A value of:<br>• SQL_OSC_MINIMUM - means that the current database management system supports minimum ODBC SQL grammar. Minimum SQL grammar must include the following elements:<br>– CREATE TABLE and DROP TABLE data definitions<br>– Simple SELECT, INSERT, UPDATE, and DELETE data manipulation<br>– Simple expressions<br>– CHAR, VARCHAR, and LONG VARCHAR data types<br>• SQL_OSC_CORE - means that the current database management system supports ODBC SQL core grammar. Core ODBC SQL grammar must include the following elements:<br>– Minimum ODBC SQL grammar<br>– ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW, GRANT, and REVOKE data definitions<br>– Full SELECT data manipulation<br>– Subquery and function expressions<br>– DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE PRECISION data types<br>• SQL_OSC_EXTENDED - means the current database management system supports extended ODBC SQL grammar. Extended ODBC SQL grammar must include the following elements:<br>– Core ODBC SQL grammar<br>– Positioned UPDATE, positioned DELETE, SELECT FOR UPDATE, and UNION data definitions<br>– Scalar functions, literal date, literal time, and literal timestamp expressions<br>– BIT, TINYINT, BIGINT, BINARY, VARBINARY, LONG VARBINARY, DATE, TIME, TIMESTAMP, and XML data types<br>– Batch SQL statements<br>– Procedure calls |
| SQL_ODBC_VER | string | The version number of ODBC that the driver manager supports.<br>Db2 ODBC returns the string "03.01.0000". |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_OJ_CAPABILITIES | 32-bit mask | A 32-bit bit mask enumerating the types of outer join supported.<br>The bit masks are:<br>• SQL_OJ_LEFT: Left outer join is supported.<br>• SQL_OJ_RIGHT: Right outer join is supported.<br>• SQL_OJ_FULL: Full outer join is supported.<br>• SQL_OJ_NESTED: Nested outer join is supported.<br>• 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.<br>• SQL_OJ_INNER: The inner table of an outer join can also be an inner join.<br>• 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'. |
| SQL_OUTER_JOINS | string | The character string:<br>• 'Y' indicates that outer joins are supported, and Db2 ODBC supports the ODBC outer join request syntax.<br>• 'N' indicates that it is not supported. |
| 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:<br>• 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.<br>• 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:<br>• 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.<br>• 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.<br>• 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. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_POSITIONED_STATEMENTS | 32-bit mask | Indicates the degree of support for positioned UPDATE and positioned DELETE statements:<br>• SQL_PS_POSITIONED_DELETE<br>• SQL_PS_POSITIONED_UPDATE<br>• 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. 'N' indicates that it does not. |
| SQL_QUOTED_IDENTIFIER_CASE | 16-bit integer | Returns:<br>• SQL_IC_UPPER - quoted identifiers in SQL are case insensitive and stored in upper case in the system catalog.<br>• SQL_IC_LOWER - quoted identifiers in SQL are case insensitive and are stored in lower case in the system catalog.<br>• SQL_IC_SENSITIVE - quoted identifiers (delimited identifiers) in SQL are case sensitive and are stored in mixed case in the system catalog.<br>• SQL_IC_MIXED - quoted identifiers in SQL are case insensitive and are stored in mixed case in the system catalog.<br>This should be contrasted with the SQL_IDENTIFIER_CASE *InfoType*, which is used to determine how (unquoted) identifiers are stored in the system catalog. |
| SQL_ROW_UPDATES | string | A character string of "Y" indicates changes are detected in rows between multiple fetches of the same rows, "N" indicates that changes are not detected. |
| SQL_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:<br>• SQL_OU_DML_STATEMENTS - supported in all Data Manipulation Language statements.<br>• SQL_OU_PROCEDURE_INVOCATION - supported in the procedure invocation statement.<br>• SQL_OU_TABLE_DEFINITION - supported in all table definition statements.<br>• SQL_OU_INDEX_DEFINITION - supported in all index definition statements.<br>• 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 options that are supported for the cursor.<br>The following bit-masks are used in conjunction with the flag to determine which attribute values are supported:<br>• SQL_SCCO_READ_ONLY<br>• SQL_SCCO_LOCK<br>• SQL_SCCO_OPT_TIMESTAMP<br>• SQL_SCCO_OPT_VALUES<br>Db2 ODBC returns SQL_SCCO_LOCK, indicating that the level of locking that is used is the lowest level of locking that is sufficient to ensure the row can be updated is used. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_SCROLL_OPTIONS | 32-bit mask | The scroll options that are supported for scrollable cursors.<br>The following bit masks are used in conjunction with the flag to determine which attribute values are supported:<br>• SQL_SO_FORWARD_ONLY<br>• SQL_SO_KEYSET_DRIVEN<br>• SQL_SO_STATIC<br>• SQL_SO_DYNAMIC<br>• SQL_SO_MIXED |
| 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 ANSI/ISO SQL standard of 1992 and that are supported in a SELECT statement.<br>• SQL_SP_BETWEEN<br>• SQL_SP_COMPARISON<br>• SQL_SP_EXISTS<br>• SQL_SP_IN<br>• SQL_SP_ISNOTNULL<br>• SQL_SP_ISNULL<br>• SQL_SP_LIKE<br>• SQL_SP_MATCH_FULL<br>• SQL_SP_MATCH_PARTIAL<br>• SQL_SP_MATCH_UNIQUE_FULL<br>• SQL_SP_MATCH_UNIQUE_PARTIAL<br>• SQL_SP_OVERLAPS<br>• SQL_SP_QUANTIFIED_COMPARISON<br>• SQL_SP_UNIQUE |
| SQL_SQL92_VALUE_EXPRESSIONS | 32-bit mask | Indicates those value expressions that are defined by SQL92 and that are supported.<br>• SQL_SVE_CASE<br>• SQL_SVE_CAST<br>• SQL_SVE_COALESCE<br>• SQL_SVE_NULLIF |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_STATIC_CURSOR_ATTRIBUTES1 | 32-bit mask | Indicates the attributes of a static cursor that Db2 ODBC supports (subset 1 of 2).<br>• SQL_CA1_NEXT<br>• SQL_CA1_ABSOLUTE<br>• SQL_CA1_RELATIVE<br>• SQL_CA1_BOOKMARK<br>• SQL_CA1_LOCK_EXCLUSIVE<br>• SQL_CA1_LOCK_NO_CHANGE<br>• SQL_CA1_LOCK_NOLOCK<br>• SQL_CA1_POS_POSITION<br>• SQL_CA1_POS_UPDATE<br>• SQL_CA1_POS_DELETE<br>• SQL_CA1_POS_REFRESH<br>• SQL_CA1_POSITIONED_UPDATE<br>• SQL_CA1_POSITIONED_DELETE<br>• SQL_CA1_SELECT_FOR_UPDATE<br>• SQL_CA1_BULK_ADD<br>• SQL_CA1_BULK_UPDATE_BY_BOOKMARK<br>• SQL_CA1_BULK_DELETE_BY_BOOKMARK<br>• SQL_CA1_BULK_FETCH_BY_BOOKMARK |
| SQL_STATIC_CURSOR_ATTRIBUTES2 | 32-bit mask | Indicates the attributes of a static cursor that Db2 ODBC supports (subset 2 of 2).<br>• SQL_CA2_READ_ONLY_CONCURRENCY<br>• SQL_CA2_LOCK_CONCURRENCY<br>• SQL_CA2_OPT_ROWVER_CONCURRENCY<br>• SQL_CA2_OPT_VALUES_CONCURRENCY<br>• SQL_CA2_SENSITIVITY_ADDITIONS<br>• SQL_CA2_SENSITIVITY_DELETIONS<br>• SQL_CA2_SENSITIVITY_UPDATES<br>• SQL_CA2_MAX_ROWS_SELECT<br>• SQL_CA2_MAX_ROWS_INSERT<br>• SQL_CA2_MAX_ROWS_DELETE<br>• SQL_CA2_MAX_ROWS_UPDATE<br>• SQL_CA2_MAX_ROWS_CATALOG<br>• SQL_CA2_MAX_ROWS_AFFECTS_ALL<br>• SQL_CA2_CRC_EXACT<br>• SQL_CA2_CRC_APPROXIMATE<br>• SQL_CA2_SIMULATE_NON_UNIQUE<br>• SQL_CA2_SIMULATE_TRY_UNIQUE<br>• SQL_CA2_SIMULATE_UNIQUE |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| 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:<br>• SQL_SS_ADDITIONS: Added rows are visible to the cursor; the cursor can scroll to these rows. All Db2 servers see added rows.<br>• 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.<br>• SQL_SS_UPDATES: Updates to rows are visible to the cursor; if the cursor scrolls from and returns to an updated row, the data returned by the cursor is the updated data, not the original data. |
| SQL_STRING_FUNCTIONS | 32-bit mask | Indicates which string functions are supported.<br>The following bit masks are used to determine which string functions are supported:<br>• SQL_FN_STR_ASCII<br>• SQL_FN_STR_CHAR<br>• SQL_FN_STR_CONCAT<br>• SQL_FN_STR_DIFFERENCE<br>• SQL_FN_STR_INSERT<br>• SQL_FN_STR_LCASE<br>• SQL_FN_STR_LEFT<br>• SQL_FN_STR_LENGTH<br>• SQL_FN_STR_LOCATE<br>• SQL_FN_STR_LOCATE_2<br>• SQL_FN_STR_LTRIM<br>• SQL_FN_STR_REPEAT<br>• SQL_FN_STR_REPLACE<br>• SQL_FN_STR_RIGHT<br>• SQL_FN_STR_RTRIM<br>• SQL_FN_STR_SOUNDEX<br>• SQL_FN_STR_SPACE<br>• SQL_FN_STR_SUBSTRING<br>• SQL_FN_STR_UCASE<br>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:<br>• SQL_SQ_COMPARISON - the *comparison* predicate<br>• SQL_SQ_CORRELATE_SUBQUERIES - all predicates<br>• SQL_SQ_EXISTS - the *exists* predicate<br>• SQL_SQ_IN - the *in* predicate<br>• SQL_SQ_QUANTIFIED - the predicates containing a quantification scalar function. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_SYSTEM_FUNCTIONS | 32-bit mask | Indicates which scalar system functions are supported.<br>The following bit masks are used to determine which scalar system functions are supported:<br>• SQL_FN_SYS_DBNAME<br>• SQL_FN_SYS_IFNULL<br>• SQL_FN_SYS_USERNAME<br>**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. |
| 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.<br>The following bit masks are used to determine which intervals are supported:<br>• SQL_FN_TSI_FRAC_SECOND<br>• SQL_FN_TSI_SECOND<br>• SQL_FN_TSI_MINUTE<br>• SQL_FN_TSI_HOUR<br>• SQL_FN_TSI_DAY<br>• SQL_FN_TSI_WEEK<br>• SQL_FN_TSI_MONTH<br>• SQL_FN_TSI_QUARTER<br>• 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.<br>The following bit masks are used to determine which intervals are supported:<br>• SQL_FN_TSI_FRAC_SECOND<br>• SQL_FN_TSI_SECOND<br>• SQL_FN_TSI_MINUTE<br>• SQL_FN_TSI_HOUR<br>• SQL_FN_TSI_DAY<br>• SQL_FN_TSI_WEEK<br>• SQL_FN_TSI_MONTH<br>• SQL_FN_TSI_QUARTER<br>• SQL_FN_TSI_YEAR |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_TIMEDATE_FUNCTIONS | 32-bit mask | Indicates which time and date functions are supported.<br>The following bit masks are used to determine which date functions are supported:<br>• SQL_FN_TD_CURDATE<br>• SQL_FN_TD_CURTIME<br>• SQL_FN_TD_DAYNAME<br>• SQL_FN_TD_DAYOFMONTH<br>• SQL_FN_TD_DAYOFWEEK<br>• SQL_FN_TD_DAYOFYEAR<br>• SQL_FN_TD_HOUR<br>• SQL_FN_TD_JULIAN_DAY<br>• SQL_FN_TD_MINUTE<br>• SQL_FN_TD_MONTH<br>• SQL_FN_TD_MONTHNAME<br>• SQL_FN_TD_NOW<br>• SQL_FN_TD_QUARTER<br>• SQL_FN_TD_SECOND<br>• SQL_FN_TD_SECONDS_SINCE_MIDNIGHT<br>• SQL_FN_TD_TIMESTAMPADD<br>• SQL_FN_TD_TIMESTAMPDIFF<br>• SQL_FN_TD_WEEK<br>• SQL_FN_TD_YEAR<br>**Tip:** These functions are intended to be used with the escape sequence in ODBC. |
| SQL_TXN_CAPABLE | 16-bit integer | Indicates whether transactions can contain Data Definition Language, or Data Manipulation Language, or both.<br>• SQL_TC_NONE - transactions not supported.<br>• SQL_TC_DML - transactions can only contain Data Manipulation Language statements (SELECT, INSERT, UPDATE, DELETE, and so on) Data Definition Language statements (CREATE TABLE, DROP INDEX, and so on) encountered in a transaction cause an error.<br>• SQL_TC_DDL_COMMIT - transactions can only contain Data Manipulation Language statements. Data Definition Language statements encountered in a transaction cause the transaction to be committed.<br>• SQL_TC_DDL_IGNORE - transactions can only contain Data Manipulation Language statements. Data Definition Language statements encountered in a transaction are ignored.<br>• SQL_TC_ALL - transactions can contain Data Definition Language and Data Manipulation Language statements in any order. |

*Table 147. Information returned by SQLGetInfo() (continued)*

| *InfoType* | Format | Description and notes |
|---|---|---|
| SQL_TXN_ISOLATION_OPTION | 32-bit mask | The transaction isolation levels available at the currently connected database server.<br>The following bit masks are used in conjunction with the flag to determine which attribute values are supported:<br>• SQL_TXN_READ_UNCOMMITTED<br>• SQL_TXN_READ_COMMITTED<br>• SQL_TXN_REPEATABLE_READ<br>• SQL_TXN_SERIALIZABLE<br>• SQL_TXN_NOCOMMIT<br>• SQL_TXN_VERSIONING<br>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:<br>• SQL_U_UNION - supports the UNION clause<br>• SQL_U_UNION_ALL - supports the ALL keyword in the UNION clause<br>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. |

### Return codes

After you call SQLGetInfo(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 148. SQLGetInfo() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | 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.) |

*Table 148. SQLGetInfo() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**003 | 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. |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**090 | Invalid string or buffer length. | The value specified for the argument *BufferLength* is less than 0. |
| **HY**096 | Invalid information type. | An invalid value is specified for the *InfoType* argument. |
| **HY**C00 | Driver not capable. | The value specified in the argument *InfoType* is not supported by Db2 ODBC or is not supported by the data source. |

**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 concepts**

Stored procedures for ODBC applications

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.

Vendor escape clauses

Vendor escape clauses increase the portability of your application if your application accesses multiple data sources from different vendors. However, if your application accesses only Db2 data sources, you have no reason to use vendor escape clauses.

**Related reference**

Changes to SQLGetInfo() InfoType argument values

Values of the *InfoType* arguments for SQLGetInfo() arguments are renamed in ODBC 3.0.

SQLGetConnectAttr() - Get current attribute setting

SQLGetConnectAttr() returns the current setting of a connection attribute and also allows you to set these attributes.

SQLGetTypeInfo() - Get data type information

SQLGetTypeInfo() returns information about the data types that are supported by the database management systems that are associated with Db2 ODBC. This information is returned in an SQL result set. The columns of this result set can be received by using the same functions that you use to process a query.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLGetLength() - Retrieve length of a string value

SQLGetLength() retrieves the length (in bytes) of a large object value. The large object value is referenced by a large object locator that the server returns. The locator can be the result of a fetch, or an SQLGetSubString() call during the current transaction.

### ODBC specifications for SQLGetLength()

*Table 149. SQLGetLength() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| No | No | No |

### Syntax

```
SQLRETURN  SQLGetLength      (SQLHSTMT          hstmt,
                              SQLSMALLINT       LocatorCType,
                              SQLINTEGER        Locator,
                              SQLINTEGER  FAR   *StringLength,
                              SQLINTEGER  FAR   *IndicatorValue);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 150. 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:<br>• SQL_C_BLOB_LOCATOR for BLOB data<br>• SQL_C_CLOB_LOCATOR for CLOB data<br>• 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 and that is not 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

**40**

*Table 151. SQLGetLength() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **07**006 | Invalid conversion. | The combination of the values that the *LocatorCType* and *Locator* arguments specify is not valid. |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **0F**001 | The LOB token variable does not currently represent any value. | The value that the *Locator* argument specifies is not associated with a LOB locator. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**003 | Program type out of range. | The *LocatorCType* argument does not specify one of the following values:<br>• SQL_C_CLOB_LOCATOR<br>• SQL_C_BLOB_LOCATOR<br>• SQL_C_DBCLOB_LOCATOR |
| **HY**009 | Invalid use of a null pointer. | The *StringLength* argument specifies a null pointer. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**C00 | 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

Refer to the function SQLGetPosition() for a related example.

### Related reference

SQLBindCol() - Bind a column to an application variable
SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLExtendedFetch() - Fetch an array of rows
SQLExtendedFetch() extends the function of SQLFetch() by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLGetPosition() - Find the starting position of a string
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.

SQLGetSubString() - Retrieve a portion of a string value
SQLGetSubString() retrieves a portion of a large object value that is referenced by a LOB locator.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLGetPosition() - Find the starting position of a string

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.

### ODBC specifications for SQLGetPosition()

*Table 152. SQLGetPosition() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| No | No | No |

### 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**

The following table lists the data type, use, and description for each argument in this function.

*Table 153. 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:<br>• SQL_C_BLOB_LOCATOR for BLOB data<br>• SQL_C_CLOB_LOCATOR for CLOB data<br>• 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.<br>This argument is ignored unless both the following conditions are met:<br>• The *SearchLiteral* argument specifies a null pointer.<br>• The *SearchLiteralLength* argument is set to 0. |
| SQLCHAR * | *SearchLiteral* | input | This argument points to the area of storage that contains the search string literal.<br>If *SearchLiteralLength* is 0, this pointer must be null. |
| SQLINTEGER | *SearchLiteralLength* | input | The length of the string in *SearchLiteral* (in bytes).[1]<br>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.<br>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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 154. SQLGetPosition() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **07**006 | Invalid conversion. | The combination of the value that the *LocatorCType* argument specifies with either of the LOB locator values is not valid. |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **0F**001 | 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. |
| **42**818 | The operands of an operator or function are not compatible. | The length of the search pattern is longer than 4000 bytes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |

*Table 154. SQLGetPosition() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**009 | Invalid use of a null pointer. | This SQLSTATE is returned for one or more of the following reasons:<br>• The pointer that the *LocatedAt* argument specifies is null.<br>• The argument value for the *FromPosition* argument is not greater than 0.<br>• The *LocatorCType* argument is not one of the following values:<br>– SQL_C_CLOB_LOCATOR<br>– SQL_C_BLOB_LOCATOR<br>– SQL_C_DBCLOB_LOCATOR |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The value of *SearchLiteralLength* is less than 1, and not SQL_NTS. |
| **HY**C00 | 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

The following example shows an application that retrieves a substring from a large object. To find 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);
/*****************************************************************
    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 position */
    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 #:
/* ... */
```

*Figure 22. An application that retrieves a substring from a large object*

**Related reference**

SQLBindCol() - Bind a column to an application variable
SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLExtendedFetch() - Fetch an array of rows
SQLExtendedFetch() extends the function of SQLFetch() by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLGetFunctions() - Get functions
SQLGetFunctions() indicates if a specific function is supported.

SQLGetLength() - Retrieve length of a string value
SQLGetLength() retrieves the length (in bytes) of a large object value. The large object value is referenced by a large object locator that the server returns. The locator can be the result of a fetch, or an SQLGetSubString() call during the current transaction.

SQLGetSubString() - Retrieve a portion of a string value
SQLGetSubString() retrieves a portion of a large object value that is referenced by a LOB locator.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLGetSQLCA() - Get SQLCA data structure

SQLGetSQLCA() returns the SQLCA (SQL communication area) that is associated with preparing and executing an SQL statement, fetching data, or closing a cursor. The SQLCA can return supplemental information about the data that is obtained by SQLGetDiagRec().

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.

### ODBC specifications for SQLGetSQLCA()

*Table 155. SQLGetSQLCA specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| No | No | No |

### Syntax

```
SQLRETURN  SQLGetSQLCA       (SQLHENV           henv,
                              SQLHDBC           hdbc,
                              SQLHSTMT          hstmt,
                              struct sqlca FAR  *pSqlca);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 156. 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 database management system, then the SQLCA contains all zeros. 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

**Example**

The following example 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;
/*****************************************************************/
/* 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;
}
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);
  }
  /*****************************************************************/
  /* 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;
  if ((rc = SQLGetSQLCA (hEnv ,
                         hDbc ,
                         hStmt,
                         pSQLCA)) != SQL_SUCCESS)
  {
    (void) printf ("**** SQLGetSQLCA failed Return Code =
    goto exit;
  }
  code = (int) pSQLCA->sqlcode;
```

```
  memcpy (state, pSQLCA->sqlstate, 5);
  state [5] = '\0';
  (void) printf ("**** sqlcode =
  memcpy (errp, pSQLCA->sqlerrp, 8);
  errp [8] = '\0';
  (void) printf ("**** sqlerrp =
  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 #
      }
    }
  }
  for (i = 0; i <= 5; i++)
    (void) printf ("**** sqlerrd #
  for (i = 0; i <= 10; i++)
    (void) printf ("**** sqlwarn #
  exit:
  return;
}
```

*Figure 23. An application that retrieves diagnostic information*

**Related reference**

SQLGetDiagRec() - Get multiple field settings of diagnostic record
SQLGetDiagRec() returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information. SQLGetDiagRec() also returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the error message text.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

Description of SQLCA fields (Db2 SQL)

# SQLGetStmtAttr() - Get current setting of a statement attribute

SQLGetStmtAttr() returns the current setting of a statement attribute. To set these statement attributes, use SQLSetStmtAttr().

**ODBC specifications for SQLGetStmtAttr()**

*Table 157. SQLGetStmtAttr() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

**Syntax**

```
SQLRETURN  SQLGetStmtAttr (SQLHSTMT          StatementHandle,
                           SQLINTEGER        Attribute,
                           SQLPOINTER        ValuePtr,
                           SQLINTEGER        BufferLength,
                           SQLINTEGER        *StringLengthPtr);
```

**Function arguments**

The following table lists the data type, use, and description for each argument in this function.

*Table 158. `SQLGetStmtAttr()` arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *StatementHandle* | input | Specifies a connection handle. |
| SQLINTEGER | *Attribute* | input | Specifies the statement attribute to retrieve. For a complete list of these attributes, refer to the function `SQLSetStmtAttr()`. |
| 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:<br>• For ODBC-defined attributes:<br>– 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.<br>– If the *ValuePtr* argument points to an integer, the *BufferLength* argument is ignored.<br>• For driver-defined attributes (IBM extension):<br>– 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).<br>– If the *ValuePtr* argument points to an integer, the *BufferLength* argument is ignored. |

*Table 158. 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.<br>• If the *ValuePtr* argument specifies a null pointer, no length is returned.<br>• If the attribute value to which *ValuePtr* points 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.<br>• 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.

**Return codes**

After you call SQLGetStmtAttr(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 159. SQLGetStmtAttr() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (SQLGetStmtAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **01**004 | 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.) |
| **HY**000 | 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. |

*Table 159. SQLGetStmtAttr() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**001 | Memory allocation failure. | Db2 ODBC can not allocate memory that is required to support execution or completion of the function. |
| **HY**010 | 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. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access memory that is required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The value specified for the *BufferLength* argument is less than 0. |
| **HY**092 | Option type out of range. | The value specified for the *Attribute* argument is not valid for this version of Db2 ODBC. |
| **HY**C00 | 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. |

**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 reference**

SQLGetConnectAttr() - Get current attribute setting
SQLGetConnectAttr() returns the current setting of a connection attribute and also allows you to set these attributes.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetConnectAttr() - Set connection attributes
SQLSetConnectAttr() sets attributes that govern aspects of connections.

SQLSetStmtAttr() - Set statement attributes

SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

## SQLGetStmtOption() - Return current setting of a statement option

SQLGetStmtOption() is a deprecated function and is replaced by SQLGetStmtAttr().

### ODBC specifications for SQLGetStmtOption()

*Table 160. SQLGetStmtOption() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | No |

### Syntax

```
SQLRETURN   SQLGetStmtOption (SQLHSTMT          hstmt,
                              SQLUSMALLINT      fOption,
                              SQLPOINTER        pvParam);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 161. 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). |

### Related reference

SQLGetStmtAttr() - Get current setting of a statement attribute
SQLGetStmtAttr() returns the current setting of a statement attribute. To set these statement attributes, use SQLSetStmtAttr().

## SQLGetSubString() - Retrieve a portion of a string value

SQLGetSubString() retrieves a portion of a large object value that is referenced by a LOB locator.

### ODBC specifications for SQLGetSubString()

*Table 162. SQLGetSubString() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| No | No | No |

### 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

The following table lists the data type, use, and description for each argument in this function.

*Table 163. 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:<br>• SQL_C_BLOB_LOCATOR for BLOB data<br>• SQL_C_CLOB_LOCATOR for CLOB data<br>• 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.<br>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 163. 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:<br>• SQL_C_CLOB_LOCATOR<br>• SQL_C_BLOB_LOCATOR<br>• SQL_C_DBCLOB_LOCATOR<br>Or, the target can be a C string variable of one of the following types:<br>• SQL_C_CHAR for CLOB data<br>• SQL_C_BINARY for BLOB data<br>• 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.<br>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 was returned by a fetch or a previous SQLGetSubString() call during the current transaction. Do not free the LOB locator through a FREE LOCATOR statement, or execute SQLGetSubString() in a different transaction from the one in which the locator is created.

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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

**40**

*Table 164. SQLGetSubString() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | Data truncated. | The amount of returned data is longer than *cbValueMax*. Actual length, in bytes, that is available for return is stored in *StringLength*. |
| **07**006 | Invalid conversion. | This SQLSTATE is returned for one or more of the following reasons:<br>• The value specified for *TargetCType* is not SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR or a LOB locator.<br>• The value specified for *TargetCType* is inappropriate for the source (for example SQL_C_DBCHAR for a BLOB column). |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **0F**001 | The LOB token variable does not currently represent any value. | The value specified for *Locator* or *SearchLocator* is not currently a LOB locator. |
| **22**011 | A substring error occurred. | *FromPosition* is greater than the length of the source string. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**003 | Program type out of range. | *LocatorCType* is not one of the following:<br>• SQL_C_CLOB_LOCATOR<br>• SQL_C_BLOB_LOCATOR<br>• SQL_C_DBCLOB_LOCATOR |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**024 | Invalid argument value. | The value specified for *FromPosition* or for *ForLength* is not a positive integer. |
| **HY**090 | Invalid string or buffer length. | The value of *cbValueMax* is less than 0. |
| **HY**C00 | 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

Refer to the function SQLGetPosition() for a related example.

### Related reference

SQLBindCol() - Bind a column to an application variable
SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLExtendedFetch() - Fetch an array of rows
SQLExtendedFetch() extends the function of SQLFetch() by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that SQLExtendedFetch() returns.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLGetLength() - Retrieve length of a string value
SQLGetLength() retrieves the length (in bytes) of a large object value. The large object value is referenced by a large object locator that the server returns. The locator can be the result of a fetch, or an SQLGetSubString() call during the current transaction.

SQLGetPosition() - Find the starting position of a string
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.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLGetTypeInfo() - Get data type information

SQLGetTypeInfo() returns information about the data types that are supported by the database management systems that are associated with Db2 ODBC. This information is returned in an SQL result set. The columns of this result set can be received by using the same functions that you use to process a query.

### ODBC specifications for SQLGetTypeInfo()

*Table 165. SQLGetTypeInfo() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLGetTypeInfo   (SQLHSTMT          hstmt,
                              SQLSMALLINT       fSqlType);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 166. SQLGetTypeInfo() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Specifies a statement handle. |
| SQLSMALLINT | *fSqlType* | input | Specifies the SQL data type that is queried. The following values that specify data types are supported:<br>• SQL_ALL_TYPES<br>• SQL_BIGINT<br>• SQL_BINARY<br>• SQL_BLOB<br>• SQL_CHAR<br>• SQL_CLOB<br>• SQL_DBCLOB<br>• SQL_DECFLOAT<br>• SQL_DECIMAL<br>• SQL_DOUBLE<br>• SQL_FLOAT<br>• SQL_GRAPHIC<br>• SQL_INTEGER<br>• SQL_LONGVARBINARY<br>• SQL_LONGVARCHAR<br>• SQL_LONGVARGRAPHIC<br>• SQL_NUMERIC<br>• SQL_REAL<br>• SQL_ROWID<br>• SQL_SMALLINT<br>• SQL_TYPE_DATE<br>• SQL_TYPE_TIME<br>• SQL_TYPE_TIMESTAMP<br>• SQL_VARBINARY<br>• SQL_VARCHAR<br>• SQL_VARGRAPHIC<br>• SQL_XML<br>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 167 on page 297 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 167. Columns returned by SQLGetTypeInfo()*

| Position | Column name | Data type | Description |
|---|---|---|---|
| 1 | TYPE_NAME | VARCHAR(128) NOT NULL | Contains a character representation of the SQL Data Definition Language 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, 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. If the data type is XML, zero is returned.<br>For date, time, timestamp data types, this is the total number of characters required to display the value when converted to characters.<br>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 suffix is not applicable. |

*Table 167. Columns returned by SQLGetTypeInfo() (continued)*

| Position | Column name | Data type | Description |
|---|---|---|---|
| 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:<br>• LENGTH, which indicates you can specify a length for the data type in the TYPE_NAME column<br>• PRECISION, which indicates you can specify the precision for the data type in the TYPE_NAME column<br>• SCALE, which indicates you can specify a scale for the data type in the TYPE_NAME column<br>• A null indicator, which indicates you cannot specify any parameters for the data type in the TYPE_NAME column<br>**Usage:** The CREATE_PARAMS column enables you to customize the interface of Data Definition Language builders in your applications. A *Data Definition Language 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 Data Definition Language builder. |
| 7 | NULLABLE | SMALLINT NOT NULL | Indicates whether the data type accepts a null value. This column contains one of the following values:<br>• SQL_NO_NULLS, which indicates that null values are disallowed<br>• 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:<br>• SQL_TRUE, which indicates case sensitivity<br>• SQL_FALSE, which indicates no case sensitivity |

*Table 167. Columns returned by SQLGetTypeInfo() (continued)*

| Position | Column name | Data type | Description |
|---|---|---|---|
| 9 | SEARCHABLE | SMALLINT NOT NULL | Indicates how the data type is used in a WHERE clause. This column contains one of the following values:<br>• SQL_UNSEARCHABLE, which indicates that you cannot use the data type in a WHERE clause<br>• SQL_LIKE_ONLY, which indicates that you can use the data type in a WHERE clause, but only with the LIKE predicate.<br>• SQL_ALL_EXCEPT_LIKE, which indicates that you can use the data type in a WHERE clause with all comparison operators except LIKE.<br>• 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:<br>• SQL_TRUE, which indicates that the data type is unsigned<br>• SQL_FALSE, which indicates the data type is signed<br>• 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.<br>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 167. 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 database management system, 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

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 168. SQLGetTypeInfo() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is open on the statement handle. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**004 | Invalid SQL data type. | An invalid value for the *fSqlType* argument is specified. |
| **HY**010 | 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 relational database management system:

**Data type**
**fSqlType**

**TINYINT**
SQL_TINYINT

**BIT**
SQL_BIT

**Example**

The following example shows an application that uses SQLGetTypeInfo() to check which ODBC data types the database management system 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;
  /************************************************************************/
  /* 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;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;
/*****************************************************************/
/* 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 24. An application that checks data types that the current server supports*

**Related reference**

SQLColAttribute() - Get column attributes
`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.

SQLExtendedFetch() - Fetch an array of rows
`SQLExtendedFetch()` extends the function of `SQLFetch()` by returning a *row set* array for each bound column. The value the SQL_ATTR_ROWSET_SIZE statement attribute determines the size of the row set that `SQLExtendedFetch()` returns.

SQLGetInfo() - Get general information
`SQLGetInfo()` returns general information about the database management systems to which the application is currently connected. For example, `SQLGetInfo()` indicates which data conversions are supported.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLMoreResults() - Check for more result sets

`SQLMoreResults()` returns more information about a statement handle. The information can be associated with an array of input parameter values for a query, or a stored procedure that returns results sets.

### ODBC specifications for SQLMoreResults()

*Table 169. SQLMoreResults() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### Syntax

```
SQLRETURN   SQLMoreResults   (SQLHSTMT          hstmt);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 170. 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 `SQLSetStmtAttr()` 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)

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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 171. SQLMoreResults() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**013 | 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.

### Restrictions

The ODBC specification of SQLMoreResults() allows row-counts that are 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**

The following example 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;
/* ... */
    /* 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: 0].Cust_Num);
        printf("************************************\n");
        while (i < pcrow)
        {   printf("\nOrder #: 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("                                j].Total);
        } /* end while */
      } /* end if */
    }
    while ( SQLMoreResults(hstmt) == SQL_SUCCESS);
/* ... */
```

*Figure 25. An application that checks for additional result sets*

**Related concepts**

Using arrays to pass parameter values
Db2 ODBC provides an array input method for updating Db2 tables.

Result sets from stored procedures in ODBC applications
In Db2 ODBC applications, you use open cursors to retrieve result sets from stored procedure calls.

**Related reference**

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator
`SQLBindParameter()` binds parameter markers to application variables and extends the capability of the `SQLSetParam()` function.

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

SQLExecDirect() - Execute a statement directly
`SQLExecDirect()` prepares and executes an SQL statement in one step.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetStmtAttr() - Set statement attributes
`SQLSetStmtAttr()` sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call `SQLSetConnectAttr()`.

# SQLNativeSql() - Get native SQL text

`SQLNativeSql()` indicates how Db2 ODBC interprets vendor escape clauses. If the original SQL string that the application passes contains vendor escape clause sequences, Db2 ODBC passes a transformed SQL string to the data source. The SQL string is passed with vendor escape clauses that are either converted or discarded.

**ODBC specifications for SQLNativeSql()**

*Table 172. SQLNativeSql() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### Syntax

```
SQLRETURN   SQLNativeSql     (SQLHDBC           hdbc,
                              SQLCHAR      FAR  *szSqlStrIn,
                              SQLINTEGER        cbSqlStrIn,
                              SQLCHAR      FAR  *szSqlStr,
                              SQLINTEGER        cbSqlStrMax,
                              SQLINTEGER   FAR  *pcbSqlStr);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 173. 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.

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 database management system 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 174. SQLNativeSql() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | 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.) |
| **08**003 | Connection is closed. | The *hdbc* argument does not reference an open database connection. |
| **37**000 | Invalid SQL syntax. | The input SQL string that the *szSqlStrIn* argument specifies contains a syntax error in the escape sequence. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**009 | Invalid use of a null pointer. | This SQLSTATE is returned for one or more of the following reasons:<br>• The argument *szSqlStrIn* is a null pointer.<br>• The argument *szSqlStr* is a null pointer. |
| **HY**090 | Invalid string or buffer length. | This SQLSTATE is returned for one or more of the following reasons:<br>• The argument *cbSqlStrIn* specifies a value that is less than 0 and not equal to SQL_NTS.<br>• The argument *cbSqlStrMax* specifies a value that is less than 0. |

**Example**

The following example 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 stmt);
       printf("Output Statement: \n stmt);
       printf("Number of Parameter Markers =
    }
    rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
/* ... */
```

*Figure 26. An application that prints a translated vendor escape clause*

**Related concepts**

Vendor escape clauses

Vendor escape clauses increase the portability of your application if your application accesses multiple data sources from different vendors. However, if your application accesses only Db2 data sources, you have no reason to use vendor escape clauses.

**Related reference**

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLNumParams() - Get number of parameters in an SQL statement

SQLNumParams() returns the number of parameter markers that are in an SQL statement.

### ODBC specifications for SQLNumParams()

*Table 175. SQLNumParams() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### Syntax

```
SQLRETURN   SQLNumParams     (SQLHSTMT          hstmt,
                              SQLSMALLINT  FAR  *pcpar);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 176. 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 177. SQLNumParams() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**009 | Invalid use of a null pointer. | The *pcpar* argument specifies a null pointer. |
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• SQLNumParams() is called before SQLPrepare() for the statement to which the *hstmt* argument refers.<br>• 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |

**Example**

Refer to the function SQLNativeSql() for a related example on an application that prints a translated vendor escape clause.

**Related reference**

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator
SQLBindParameter() binds parameter markers to application variables and extends the capability of the SQLSetParam() function.

SQLNativeSql() - Get native SQL text
SQLNativeSql() indicates how Db2 ODBC interprets vendor escape clauses. If the original SQL string that the application passes contains vendor escape clause sequences, Db2 ODBC passes a transformed SQL string to the data source. The SQL string is passed with vendor escape clauses that are either converted or discarded.

SQLPrepare() - Prepare a statement
SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLNumResultCols() - Get number of result columns

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 you call

SQLNumResultCols(). After you call SQLNumResultCols(), you can call SQLColAttribute() or one of the bind column functions.

### ODBC specifications for SQLNumResultCols()

*Table 178. SQLNumResultCols() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLNumResultCols (SQLHSTMT          hstmt,
                              SQLSMALLINT FAR   *pccol);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 179. 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 180. SQLNumResultCols() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |

*Table 180. SQLNumResultCols() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**009 | Invalid use of a null pointer. | *pccol* is a null pointer. |
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• The function is called prior to calling SQLPrepare() or SQLExecDirect() for the *hstmt*.<br>• 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |

**Example**

Refer to the function SQLDescribeCol() for a related example.

**Related reference**

SQLBindCol() - Bind a column to an application variable
SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLColAttribute() - Get column attributes
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.

SQLDescribeCol() - Describe column attributes
SQLDescribeCol() returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

SQLExecDirect() - Execute a statement directly
SQLExecDirect() prepares and executes an SQL statement in one step.

SQLGetData() - Get data from a column
SQLGetData() retrieves data for a single column in the current row of the result set. You can also use SQLGetData() to retrieve large data values in pieces. After you call SQLGetData() for each column, call SQLFetch() or SQLExtendedFetch() for each row that you want to retrieve.

SQLPrepare() - Prepare a statement
SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLParamData() - Get next parameter for which a data value is needed

SQLParamData() is used in conjunction with SQLPutData() to send long data in pieces. You can also use this function to send fixed-length data.

### ODBC specifications for SQLParamData()

*Table 181. SQLParamData() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLParamData     (SQLHSTMT          hstmt,
                              SQLPOINTER  FAR   *prgbValue);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 182. 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

### Diagnostics

SQLParamData() can return any SQLSTATE that SQLExecDirect() and SQLExecute() generate. The following table lists the additional SQLSTATEs that SQLParamData() can generate with a description and explanation for each value.

*Table 183. SQLParamData() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **40**001 | Transaction rollback. | The transaction to which this SQL statement belongs is rolled back due to a deadlock or timeout. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• SQLParamData() is called out of sequence. This call is only valid after an SQLExecDirect() or an SQLExecute(), or after an SQLPutData() call.<br>• SQLParamData() is called after an SQLExecDirect() or an SQLExecute() call, but no SQL_DATA_AT_EXEC parameters require processing. |

### Example

Refer to the function SQLGetData() for a related example.

### Related reference

SQLBindCol() - Bind a column to an application variable
SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLColAttribute() - Get column attributes
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.

SQLDescribeCol() - Describe column attributes

SQLDescribeCol() returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

SQLExecDirect() - Execute a statement directly
SQLExecDirect() prepares and executes an SQL statement in one step.

SQLGetData() - Get data from a column
SQLGetData() retrieves data for a single column in the current row of the result set. You can also use SQLGetData() to retrieve large data values in pieces. After you call SQLGetData() for each column, call SQLFetch() or SQLExtendedFetch() for each row that you want to retrieve.

SQLPrepare() - Prepare a statement
SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLParamOptions() - Specify an input array for a parameter

SQLParamOptions() is a deprecated function and is replaced by SQLSetStmtAttr().

### ODBC specifications for SQLParamOptions()

*Table 184. SQLParamOptions() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | No | No |

### Syntax

```
SQLRETURN   SQLParamOptions  (SQLHSTMT          hstmt,
                              SQLUINTEGER       crow,
                              SQLUINTEGER  FAR  *pirow);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 185. 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() returns 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.

If the statement to which SQLParamOptions() refers is a MERGE statement:

- Use SQLParamOptions() to set the number of rows in the source data to be merged into the target table or view.
- If a MERGE statement contains an UPDATE or INSERT clause with parameter markers, SQLParamOptions() has no effect on the parameter markers in the UPDATE or INSERT clause.
- The buffer to which *pirow* points contains the number of rows that are affected by the MERGE.

### Return codes

After you call SQLParamOptions(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 186. SQLParamOptions() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**107 | Row value out of range. | The value in the *crow* argument is less than 1. |

**Example**

In ODBC 3.0, the call to SQLParamOptions() is replaced with two calls to SQLSetStmtAttr():

```
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, crow, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, piRow, 0);
```

**Related concepts**

Using arrays to pass parameter values
Db2 ODBC provides an array input method for updating Db2 tables.

**Related reference**

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator
`SQLBindParameter()` binds parameter markers to application variables and extends the capability of the `SQLSetParam()` function.

SQLMoreResults() - Check for more result sets
`SQLMoreResults()` returns more information about a statement handle. The information can be associated with an array of input parameter values for a query, or a stored procedure that returns results sets.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetStmtAttr() - Set statement attributes
`SQLSetStmtAttr()` sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call `SQLSetConnectAttr()`.

# SQLPrepare() - Prepare a statement

`SQLPrepare()` associates an SQL statement with the input statement handle and sends the statement to the database management system where it is 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 `SQLPrepare()` is called.

**ODBC specifications for SQLPrepare()**

*Table 187. SQLPrepare() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

**Syntax**

```
SQLRETURN   SQLPrepare       (SQLHSTMT          hstmt,
                              SQLCHAR      FAR  *szSqlStr,
                              SQLINTEGER        cbSqlStr);
```

**Function arguments**

The following table lists the data type, use, and description for each argument in this function.

*Table 188. 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.<br>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, 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().

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 for z/OS supports, see the topic *Differences between Db2 ODBC and embedded SQL*.

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.

If the statement that is being prepared is a MERGE statement, the statement text cannot include the FOR *n* ROWS clause. To specify the number of rows to be merged, use the SQLSetStmtAttr() function.

**Return codes**

After you call SQLPrepare(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 189. SQLPrepare() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**504 | 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. |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **21**S01 | Insert value list does not match column list. | *szSqlStr* contains an INSERT or MERGE statement and the number of values to be inserted did not match the degree of the derived table. |
| **21**S02 | 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. |
| **24**000 | Invalid cursor state. | A cursor is already opened on the statement handle. |
| **34**000 | Invalid cursor name. | *szSqlStr* contains a positioned DELETE or a positioned UPDATE and the cursor referenced by the statement being executed is not open. |
| **37***xxx*[1] | Invalid SQL syntax. | *szSqlStr* contains one or more of the following:<br>• A COMMIT<br>• A ROLLBACK<br>• An SQL statement that the connected database server cannot prepare<br>• A statement containing a syntax error |
| **40**001 | Transaction rollback. | The transaction to which this SQL statement belongs is rolled back due to deadlock or timeout. |
| **42***xxx* [1] | Syntax error or access rule violation | These SQLSTATEs indicate one of the following errors:<br>• For **42**5*xx*, the authorization ID does not have permission to execute the SQL statement that the *szSqlStr* argument contains.<br>• For **42***xxx*, a variety of syntax or access problems with the statement occur. |
| **42**S01 | Database object already exists. | *szSqlStr* contains a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already exists. |
| **42**S02 | Database object does not exist. | *szSqlStr* contains an SQL statement that references a table name or a view name that does not exist. |
| **42**S11 | Index already exists. | *szSqlStr* contains a CREATE INDEX statement and the specified index name already exists. |
| **42**S12 | Index not found. | *szSqlStr* contains a DROP INDEX statement and the specified index name does not exist. |

*Table 189. SQLPrepare() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **42**S21 | 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. |
| **42**S22 | Column not found. | *szSqlStr* contains an SQL statement that references a column name that does not exist. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**009 | Invalid use of a null pointer. | *szSqlStr* is a null pointer. |
| **HY**010 | 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |
| **HY**090 | 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 database management systems 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

If the statement that is being prepared is a MERGE statement, the statement text cannot include the FOR *n* ROWS clause. To specify the number of rows to be merged, use the SQLSetStmtAttr() function with the SQL_ATTR_PARAMSET_SIZE statement attribute.

### Example

The following example 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;
/*****************************************************************/
/* 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)
{
  (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)
  {
    (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;
  /*****************************************************************/
  /* 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 27. An application that prepares an SQL statement before execution*

**Related concepts**

Differences between Db2 ODBC and embedded SQL
Even though key differences exist between Db2 ODBC and embedded SQL, Db2 ODBC can execute any SQL statements that can be prepared dynamically in embedded SQL.

Vendor escape clauses
Vendor escape clauses increase the portability of your application if your application accesses multiple data sources from different vendors. However, if your application accesses only Db2 data sources, you have no reason to use vendor escape clauses.

**Related reference**

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator
`SQLBindParameter()` binds parameter markers to application variables and extends the capability of the `SQLSetParam()` function.

SQLColAttribute() - Get column attributes
`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.

SQLDescribeCol() - Describe column attributes
`SQLDescribeCol()` returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either `SQLPrepare()` or `SQLExecDirect()`.

SQLExecDirect() - Execute a statement directly
`SQLExecDirect()` prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
`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.

SQLNumParams() - Get number of parameters in an SQL statement
`SQLNumParams()` returns the number of parameter markers that are in an SQL statement.

SQLNumResultCols() - Get number of result columns
`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 you call `SQLNumResultCols()`. After you call `SQLNumResultCols()`, you can call `SQLColAttribute()` or one of the bind column functions.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetParam() - Bind a parameter marker to a buffer
`SQLSetParam()` is a deprecated function and is replaced with `SQLBindParameter()`.

# SQLPrimaryKeys() - Get primary key columns of a table

`SQLPrimaryKeys()` returns a list of column names that comprise the primary key for a table. The information is returned in an SQL result set. This result set can be retrieved by using the same functions that process a result set that is generated by a query.

**ODBC specifications for SQLPrimaryKeys()**

*Table 190. `SQLPrimaryKeys()` specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### Syntax

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

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 191. SQLPrimaryKeys() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Statement handle. |
| SQLCHAR * | *szCatalogName* | input | Catalog qualifier of a three-part table name.<br>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 192 on page 325, 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 ANSI/ISO SQL standard of 1992 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 database management system supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected database management system supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected database management system supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected database management system 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. The following table lists each column in the result set this function generates.

*Table 192. Columns returned by SQLPrimaryKeys()*

| Column number | Column 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

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 193. SQLPrimaryKeys() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **24**000 | Invalid cursor state. | A cursor is already open on the statement handle. |
| **40**003 or **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |
| **HY**090 | Invalid string or buffer length. | The value of one of the name length arguments is less than 0, but not equal SQL_NTS. |
| **HY**C00 | Driver not capable. | Db2 ODBC does not support *catalog* as a qualifier for table name. |

**Example**

The following example 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);
   }

   /*****************************************************************/
   /*  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 ==>
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 28. An application that locates a table's primary key*

**Related reference**

SQLForeignKeys() - Get a list of foreign key columns
`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 that is generated by a query.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLStatistics() - Get index and statistics information for a base table
`SQLStatistics()` retrieves index information for a specific table. It also returns the cardinality and the number of pages that are associated with the table and the indexes on the table. The information is returned in a result set. You can retrieve the result set with the same functions that process a result set that is generated by a query.

# SQLProcedureColumns() - Get procedure input/output parameter information

`SQLProcedureColumns()` returns a list of input and output parameters that are associated with a procedure. The information is returned in an SQL result set. This result set is retrieved by the same functions that process a result set that is generated by a query.

### ODBC specifications for SQLProcedureColumns()

*Table 194. SQLProcedureColumns() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### 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**

The following table lists the data type, use, and description for each argument in this function.

*Table 195. SQLProcedureColumns() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Statement handle. |
| SQLCHAR * | *szProcCatalog* | input | Catalog qualifier of a three-part procedure name.<br>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.<br>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.<br>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*. |
| 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*.<br>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*. |

**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 196 on page 329 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 ANSI/ISO SQL standard of 1992 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 database management system supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected database management system supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected database management system supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected database management system 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. The following table lists these columns.

*Table 196. Columns returned by SQLProcedureColumns()*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 1 | PROCEDURE_CAT | VARCHAR(128) | This field is always null. |
| 2 | PROCEDURE_SCHEM | VARCHAR(128) | The name of the schema containing PROCEDURE_NAME. (This is also NULL for Db2 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:<br>• SQL_PARAM_TYPE_UNKNOWN: the parameter type is unknown.[1]<br>• SQL_PARAM_INPUT: this parameter is an input parameter.<br>• SQL_PARAM_INPUT_OUTPUT: this parameter is an input/output parameter.<br>• SQL_PARAM_OUTPUT: this parameter is an output parameter.<br>• SQL_RETURN_VALUE: the procedure column is the return value of the procedure.[1]<br>• SQL_RESULT_COL: this parameter is actually a column in the result set."1" on page 332<br>**Requirement:** For SQL_PARAM_OUTPUT and SQL_RETURN_VALUE support, you must have ODBC 2.0 or higher. |
| 6 | DATA_TYPE | SMALLINT NOT NULL | SQL data type. |

*Table 196. Columns returned by SQLProcedureColumns() (continued)*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 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.<br><br>For date, time, timestamp data types, this is the total number of bytes required to display the value when converted to character.<br><br>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.<br><br>For the XML data type in native SQL procedures, zero is returned (the XML data type has no length). |
| 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.<br><br>For the XML data type in native SQL procedures, zero is returned (the XML data type has no length). |
| 10 | DECIMAL_DIGITS | SMALLINT | The scale of the parameter. NULL is returned for data types where scale is not applicable. |
| 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.<br><br>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.<br><br>For numeric data types, the database management system can return a NUM_PREC_RADIX of either 10 or 2.<br><br>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.<br><br>SQL_NULLABLE if the parameter accepts NULL values. |

*Table 196. Columns returned by SQLProcedureColumns() (continued)*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 13 | REMARKS | VARCHAR(254) | Might contain descriptive information about the parameter. |
| 14 | COLUMN_DEF | VARCHAR(254) | The default value for the column. If the default value is:<br>• A numeric literal, this column contains the character representation of the numeric literal with no enclosing single quotes.<br>• A character string, this column is that string enclosed in single quotes.<br>• 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.<br>• NULL, this column returns the word NULL, with no enclosing single quotes.<br>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:<br>• SQL_CODE_DATE<br>• SQL_CODE_TIME<br>• SQL_CODE_TIMESTAMP<br>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 the XML data type in native SQL procedures, zero is returned (the XML data type has no length). For all other data types, this column returns null value. |
| 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. |

*Table 196. Columns returned by SQLProcedureColumns() (continued)*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 19 | IS_NULLABLE | VARCHAR(128) | One of the following:<br>• "NO", if the column does not include null values<br>• "YES", if the column can include null values<br>• Zero-length string if nullability is unknown.<br>The value returned for this column is different than the value returned for the NULLABLE column. (See the description of the NULLABLE column.) |

**Note:**

1. These values are not returned.

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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 197. SQLProcedureColumns() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is opened on the statement handle. |
| **42**601 | PARMLIST syntax error. | The PARMLIST value in the stored procedures catalog table contains a syntax error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |
| **HY**090 | Invalid string or buffer length | The value of one of the name length arguments is less than 0, but not equal SQL_NTS. |

*Table 197. SQLProcedureColumns() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**C00 | Driver not capable. | This SQLSTATE is returned for one or more of the following reasons:<br>• Db2 ODBC does not support *catalog* as a qualifier for procedure name.<br>• 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

The following example 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);

   (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);

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;
```

```
  }
  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;
  /*****************************************************************/
  /* 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 29. An application that retrieves parameters associated with a procedure*

**Related concepts**

Use of ODBC for querying the Db2 catalog

You can use Db2 ODBC catalog query functions and direct catalog queries to the Db2 ODBC shadow catalog to obtain catalog information.

**Related reference**

Length of SQL data types
The length of a column is the maximum number of bytes that are returned to the application when data is transferred to its default C data type.

Precision of SQL data types
The precision of a numeric column or parameter refers to the maximum number of digits that are used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter.

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.

SQLProcedures() - Get a list of procedure names
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. This result set is retrieved by the same functions that process a result set that is generated by a query.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLProcedures() - Get a list of procedure names

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. This result set is retrieved by the same functions that process a result set that is generated by a query.

### ODBC specifications for SQLProcedures()

*Table 198. SQLProcedures() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### Syntax

```
SQLRETURN   SQLProcedures    (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szProcCatalog,
                              SQLSMALLINT       cbProcCatalog,
                              SQLCHAR     FAR   *szProcSchema,
                              SQLSMALLINT       cbProcSchema,
                              SQLCHAR     FAR   *szProcName,
                              SQLSMALLINT       cbProcName);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 199. SQLProcedures() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Statement handle. |

*Table 199. SQLProcedures() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLCHAR * | *szProcCatalog* | input | Catalog qualifier of a three-part procedure name.<br>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.<br>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.<br>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*. |

**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 200 on page 338 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 ANSI/ISO SQL standard of 1992 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 database management system supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected database management system supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected database management system supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected database management system 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 201 on page 338 lists these columns.

*Table 200. 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. |
| 8 | PROCEDURE_TYPE | SMALLINT | Defines the procedure type:<br>• SQL_PT_UNKNOWN: It cannot be determined whether the procedure returns a value.<br>• SQL_PT_PROCEDURE: The returned object is a procedure; that is, it does not have a return value.<br>• SQL_PT_FUNCTION: The returned object is a function; that is, it has a return value.<br>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

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 201. SQLProcedures() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is opened on the statement handle. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |

*Table 201. SQLProcedures() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**010 | 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.) |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |
| **HY**090 | Invalid string or buffer length. | The value of one of the name length arguments is less than 0, but not equal to SQL_NTS. |
| **HY**C00 | Driver not capable. | This SQLSTATE is returned for one or more of the following reasons:<br>• Db2 ODBC does not support *catalog* as a qualifier for procedure name.<br>• 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

The following example shows an application that prints a list of procedures registered at the server. The application uses SQLProcedures() to 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, "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("schem.s, proc_name.s);
        if (remarks.ind != SQL_NULL_DATA) {
            printf("  (Remarks)
        }
    }                           /* endwhile */
/* ... */
```

*Figure 30. An application that prints a list of registered procedures*

### Related concepts

Use of ODBC for querying the Db2 catalog

You can use Db2 ODBC catalog query functions and direct catalog queries to the Db2 ODBC shadow catalog to obtain catalog information.

### Related reference

SQLProcedureColumns() - Get procedure input/output parameter information

SQLProcedureColumns() returns a list of input and output parameters that are associated with a procedure. The information is returned in an SQL result set. This result set is retrieved by the same functions that process a result set that is generated by a query.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLPutData() - Pass a data value for a parameter

SQLPutData() supplies 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. This result set is retrieved by the same functions that process a result set that is generated by a query.

### ODBC specifications for SQLPutData()

*Table 202. SQLPutData() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

For 31-bit applications, use the following syntax:

```
SQLRETURN   SQLPutData       (SQLHSTMT          hstmt,
                              SQLPOINTER        rgbValue,
                              SQLINTEGER        cbValue);
```

For 64-bit applications, use the following syntax:

```
SQLRETURN   SQLPutData       (SQLHSTMT          hstmt,
                              SQLPOINTER        rgbValue,
                              SQLLEN            cbValue);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 203. 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. |

*Table 203. SQLPutData() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLINTEGER (31-bit) or SQLLEN (64-bit) [1] | *cbValue* | input | The length, in bytes, of *rgbValue*. Specifies the amount of data sent in a call to SQLPutData().<br><br>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*.<br><br>*cbValue* is ignored for all fixed-length C buffer types, such as date, time, timestamp, and all numeric C buffer types.<br><br>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. |

**Notes:**

1. For 64-bit applications, the data type SQLINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLLEN is recommended.

### Usage

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

**Diagnostics**

Some of the following diagnostic conditions are also reported on the final SQLParamData() call rather than at the time the SQLPutData() is called. The following table lists each SQLSTATE with a description and explanation for each value.

*Table 204. SQLPutData() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | Data truncated. | This SQLSTATE is returned for one or more of the following reasons:<br>• The data sent for a numeric parameter is truncated without the loss of significant digits.<br>• Timestamp data sent for a date or time column is truncated.<br>(SQLPutData() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **22**001 | String data right truncation. | More data is sent for a binary or char data than the data source can support for that column. |
| **22**008 | Invalid datetime format or datetime field overflow. | The data value sent for a date, time, or timestamp parameters is invalid. |
| **22**018 | Error in assignment. | The data sent for a parameter is incompatible with the data type of the associated table column. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**009 | Invalid use of a null pointer. | The argument *rgbValue* is a NULL pointer, and the argument *cbValue* is neither 0 nor SQL_NULL_DATA. |
| **HY**010 | 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. |
| **HY**019 | Numeric value out of range. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• SQLPutData() is called more than once for a fixed-length parameter. |
| **HY**090 | 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**

Refer to the function SQLGetData() for a related example.

**Related concepts**

Input and retrieval of long data in pieces
When an application must manipulate long data values, loading the entire values into storage can become impractical. For this reason, Db2 ODBC provides a technique that enables you to handle long data values in pieces.

**Related reference**

SQLBindParameter() - Bind a parameter marker to a buffer or LOB locator
SQLBindParameter() binds parameter markers to application variables and extends the capability of the SQLSetParam() function.

SQLCancel() - Cancel statement
SQLCancel() terminates an SQLExecDirect() or SQLExecute() sequence prematurely.

SQLExecDirect() - Execute a statement directly
SQLExecDirect() prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
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.

SQLGetData() - Get data from a column
SQLGetData() retrieves data for a single column in the current row of the result set. You can also use SQLGetData() to retrieve large data values in pieces. After you call SQLGetData() for each column, call SQLFetch() or SQLExtendedFetch() for each row that you want to retrieve.

SQLParamData() - Get next parameter for which a data value is needed
SQLParamData() is used in conjunction with SQLPutData() to send long data in pieces. You can also use this function to send fixed-length data.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLRowCount() - Get row count

SQLRowCount() returns the number of rows in a table that were affected by an UPDATE, INSERT, DELETE, or MERGE statement. You can call SQLRowCount() against a table or against a view that is

based on the table. SQLExecute() or SQLExecDirect() must be called before SQLRowCount() is called.

### ODBC specifications for SQLRowCount()

*Table 205. SQLRowCount() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

For 31-bit applications, use the following syntax:

```
SQLRETURN   SQLRowCount      (SQLHSTMT          hstmt,
                              SQLINTEGER FAR   *pcrow);
```

For 64-bit applications, use the following syntax:

```
SQLRETURN   SQLRowCount      (SQLHSTMT          hstmt,
                              SQLLEN     FAR   *pcrow);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 206. SQLRowCount() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Statement handle |
| SQLINTEGER *(31-bit) or SQLLEN * (64-bit)[1] | *pcrow* | output | Pointer to location where the number of rows affected is stored. |

**Notes:**

1. For 64-bit applications, the data type SQLINTEGER, which was used in previous versions of Db2, is still valid. However, for maximum application portability, using SQLLEN is recommended.

### Usage

If the last executed statement referenced by the input statement handle is not an UPDATE, INSERT, DELETE, or MERGE 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, DELETE, or MERGE, 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

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 207. SQLRowCount() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | Function sequence error. | The function is called prior to calling SQLExecute() or SQLExecDirect() for the *hstmt*. |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |

**Example**

Refer to the function SQLDescribeCol() for a related example.

**Related reference**

SQLDescribeCol() - Describe column attributes
SQLDescribeCol() returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

SQLExecDirect() - Execute a statement directly
SQLExecDirect() prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
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.

SQLNumResultCols() - Get number of result columns
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 you call SQLNumResultCols(). After you call SQLNumResultCols(), you can call SQLColAttribute() or one of the bind column functions.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLSetColAttributes() - Set column attributes

SQLSetColAttributes() sets the data source result descriptor (column name, type, precision, scale, and nullability) for one column in the result set. If you set the data source result descriptor, Db2 ODBC does not need to obtain the descriptor information from the database management system server.

### ODBC specifications for SQLSetColAttributes()

*Table 208. SQLSetColAttributes() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| No | No | No |

### Syntax

```
SQLRETURN  SQLSetColAttributes (SQLHSTMT        hstmt,
                                SQLUSMALLINT    icol,
                                SQLCHAR     FAR *pszColName,
                                SQLSMALLINT     cbColName,
                                SQLSMALLINT     fSqlType,
                                SQLUINTEGER     cbColDef,
                                SQLSMALLINT     ibScale,
                                SQLSMALLINT     fNullable);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 209. 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 209. SQLSetColAttributes() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLSMALLINT | *fSqlType* | input | The SQL data type of the column. The following values are recognized:<br>• SQL_BIGINT<br>• SQL_BINARY<br>• SQL_BLOB<br>• SQL_CHAR<br>• SQL_CLOB<br>• SQL_DBCLOB<br>• SQL_DECFLOAT<br>• SQL_DECIMAL<br>• SQL_DOUBLE<br>• SQL_FLOAT<br>• SQL_GRAPHIC<br>• SQL_INTEGER<br>• SQL_LONGVARBINARY<br>• SQL_LONGVARCHAR<br>• SQL_LONGVARGRAPHIC<br>• SQL_NUMERIC<br>• SQL_REAL<br>• SQL_ROWID<br>• SQL_SMALLINT<br>• SQL_TYPE_DATE<br>• SQL_TYPE_TIME<br>• SQL_TYPE_TIMESTAMP<br>• SQL_VARBINARY<br>• SQL_VARCHAR<br>• SQL_VARGRAPHIC<br>• SQL_XML |
| SQLUINTEGER | *cbColDef* | input | The precision of the column on the data source. If *fSqlType* is SQL_XML, ODBC ignores *cbColDef*. |
| 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:<br>• SQL_NO_NULLS - the column does not allow null values.<br>• 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

#### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 210. SQLSetColAttributes() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**004 | 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. |
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is open on the statement handle. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**004 | Invalid SQL data type. | The value specified for the argument *fSqlType* is not a valid SQL data type. |
| **HY**010 | 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.) |

*Table 210. SQLSetColAttributes() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The value specified for the argument *cbColName* is less than 0 and not equal to SQL_NTS. |
| **HY**099 | Nullable type out of range. | The value specified for *fNullable* is invalid. |
| **HY**104 | Invalid precision value. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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).<br>• The value specified for *fSqlType* is SQL_TYPE_TIMESTAMP and the value for *ibScale* is less than 0 or greater than 12. |
| **HY**002 | 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. |

**Example**

The following example 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 31. An application that sets the data source results descriptor*

**Related reference**

SQLColAttribute() - Get column attributes
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.

SQLDescribeCol() - Describe column attributes

SQLDescribeCol() returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

SQLExecDirect() - Execute a statement directly
SQLExecDirect() prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
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.

SQLPrepare() - Prepare a statement
SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

## SQLSetConnectAttr() - Set connection attributes

SQLSetConnectAttr() sets attributes that govern aspects of connections.

### ODBC specifications for SQLSetConnectAttr()

*Table 211. SQLSetConnectAttr() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

```
SQLRETURN  SQLSetConnectAttr (SQLHDBC           ConnectionHandle,
                              SQLINTEGER        Attribute,
                              SQLPOINTER        ValuePtr,
                              SQLINTEGER        StringLength);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 212. SQLSetConnectAttr() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHDBC | *ConnectionHandle* | input | Connection handle. |
| SQLINTEGER | *Attribute* | input | Connection attribute to set. Refer to Table 214 on page 352 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. |

*Table 212. SQLSetConnectAttr() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLINTEGER | *StringLength* | input | Information about the **ValuePtr* argument.<br>• For ODBC-defined attributes:<br>– If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*.<br>– If *ValuePtr* points to an integer, *BufferLength* is ignored.<br>• For driver-defined attributes (IBM extension):<br>– 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.<br>– If *ValuePtr* points to an integer, *BufferLength* is ignored. |

**Usage**

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. The following table indicates when each of the connection attributes can be set.

*Table 213. When connection attributes can be set*

| ***Attribute*** | **Before connection** | **After connection** | **After statements are allocated** |
|---|---|---|---|
| SQL_ATTR_ACCESS_MODE | Yes | Yes | Yes"1" on page 351 |
| SQL_ATTR_AUTOCOMMIT | Yes | Yes | Yes"2" on page 351 |
| SQL_ATTR_CONCURRENT_ACCESS_RESOLUTION | Yes | Yes | Yes |
| SQL_ATTR_CONNECTTYPE | Yes | No | No |
| SQL_ATTR_CURRENT_SCHEMA | Yes | Yes | Yes |
| SQL_ATTR_DB2EXPLAIN | Yes | Yes | Yes |
| SQL_ATTR_DECFLOAT_ROUNDING_MODE | Yes | Yes | Yes |
| SQL_ATTR_EXTENDED_INDICATORS | 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 214 on page 352 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, refer to "Changes to SQLSetConnectAttr() attributes".

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 214. Connection attributes*

| ***Attribute*** | ***ValuePtr*** |
|---|---|
| SQL_ATTR_ACCESS_MODE | A 32-bit integer value which can be either:<br>**SQL_MODE_READ_ONLY**<br>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).<br>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.<br>**QL_MODE_READ_WRITE**<br>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.<br>SQL_MODE_READ_WRITE is the default.<br>This connection must have no outstanding transactions. |

*Table 214. Connection attributes (continued)*

| ***Attribute*** | ***ValuePtr*** |
|---|---|
| SQL_ATTR_AUTOCOMMIT[1] | A 32-bit integer value that specifies whether to use autocommit or manual commit mode:<br>**SQL_AUTOCOMMIT_OFF**<br>The application must manually, explicitly commit or rollback transactions with `SQLEndTran()` calls.<br>**SQL_AUTOCOMMIT_ON**<br>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.<br>**Exception:** If the connection is a coordinated distributed unit of work connection, the default is **SQL_AUTOCOMMIT_OFF**.<br>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.<br>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.<br>Changing from manual-commit to autocommit mode commits any open transaction on the connection.<br>For information about setting this attribute see the topic *Disable autocommit to reduce network flow*. |

*Table 214. Connection attributes (continued)*

| ***Attribute*** | ***ValuePtr*** |
|---|---|
| SQL_ATTR_CONCURRENT_ ACCESS_RESOLUTION | A 32-bit integer value that specifies how the application resolves concurrent access to locked data:<br>**0**<br>No setting. This is the default value.<br>**1**<br>USE CURRENTLY COMMITTED. Read transactions can access the currently committed version of the data when the data is in the process of being updated or deleted. Rows that are in the process of being inserted are skipped. After this value is set through SQLSetConnectAttr(), all user SELECT statements for that connection are prepared with the USE CURRENTLY COMMITTED attribute. This option applies only when cursor stability (CS) isolation is in effect.<br>**2**<br>WAIT FOR OUTCOME. Read transactions that require access to data that is in the process of being updated or deleted must wait for a COMMIT or ROLLBACK operation to complete. Rows in the process of being inserted are not skipped. After this value is set through SQLSetConnectAttr(), all user SELECT statements for that connection are prepared with the WAIT FOR OUTCOME attribute.<br>**3**<br>SKIP LOCKED DATA. Read transactions can skip any rows that are incompatibly locked by other transactions. After this value is set through SQLSetConnectAttr(), all user SELECT statements for that connection are prepared with the SKIP LOCKED DATA attribute. This option applies only when cursor stability or read stability (RS) or cursor stability (CS) isolation are in effect.<br>SQL_ATTR_CONCURRENT_ACCESS_RESOLUTION can be set before or after a connection is made. It can also be set after statements are allocated, however it will only affect subsequently allocated statements. |

*Table 214. 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:<br>**SQL_CONCURRENT_TRANS**<br>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.<br>The current setting of the SQL_ATTR_SYNC_POINT attribute is ignored.<br>This is the default.<br>**SQL_COORDINATED_TRANS**<br>The application wants 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.<br>**Important:** This connection type results in the default for SQL_ATTR_AUTOCOMMIT connection attribute to be SQL_AUTOCOMMIT_OFF.<br>This attribute must be set before making a connect request; otherwise, the `SQLSetConnectAttr()` call is rejected.<br>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.<br>**IBM specific:** This attribute is an IBM-defined extension.<br>**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. |

*Table 214. Connection attributes (continued)*

| ***Attribute*** | ***ValuePtr*** |
|---|---|
| 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.<br><br>To reset this attribute, specify this attribute with a zero length or a null pointer for the *vParam* argument.<br><br>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.<br><br>This attribute can be set at any time and is effective on the next `SQLColumns()` call where the *szSchemaName* pointer is null.<br><br>**IBM specific:** This attribute is an IBM-defined extension. |
| SQL_ATTR_DB2EXPLAIN | A 32-bit integer value that specifies whether EXPLAIN information should be gathered. This attribute sets the CURRENT EXPLAIN MODE special register. You can specify one of the following values:<br><br>**SQL_DB2EXPLAIN_OFF**<br>Sets the CURRENT EXPLAIN MODE special register to NO, which disables the EXPLAIN facility.<br><br>**SQL_DB2EXPLAIN_MODE_ON**<br>Sets the CURRENT EXPLAIN MODE special register to YES, which enables the EXPLAIN facility.<br><br>If you enable the EXPLAIN facility, you must meet the following requirements:<br><br>• The EXPLAIN tables must exist.<br>• The current authorization ID must have INSERT privilege for the EXPLAIN tables.<br><br>The new SQL_ATTR_DB2EXPLAIN setting is effective on the next statement preparation for this connection.<br><br>Alternatively, you can set the CURRENT EXPLAIN MODE special register for ODBC applications by using the DB2EXPLAIN initialization keyword or the SET CURRENT EXPLAIN MODE SQL statement. If you want to set the CURRENT EXPLAIN MODE special register to EXPLAIN, you must use the SET CURRENT EXPLAIN MODE SQL statement.<br><br>**IBM specific:** This attribute is an IBM-defined extension. |

*Table 214. Connection attributes (continued)*

| *Attribute* | *ValuePtr* |
|---|---|
| SQL_ATTR_DECFLOAT_ ROUNDING_MODE | A 32-bit integer value that lets an application control the rounding mode for DECFLOAT data type values. Possible values are:<br>**ROUND_HALF_EVEN**<br>Round to the nearest integer. If the value is equidistant from two integers, round so that the final digit is even.<br>**ROUND_HALF_UP**<br>Round to the nearest integer. If the value is equidistant from two integers, round up.<br>**ROUND_DOWN**<br>Round toward 0. This is equivalent to truncation.<br>**ROUND_CEILING**<br>Round toward positive infinity.<br>**ROUND_FLOOR**<br>Round toward negative infinity.<br>**ROUND_HALF_DOWN**<br>Round to the nearest integer. If the value is equidistant from two integers, round down.<br>**ROUND_UP**<br>Round away from zero. |
| SQL_ATTR_EXTENDED_INDICATORS | A 32-bit integer value that overrides the EXTENDEDINDICATOR initialization keyword value.<br>**SQL_TRUE**<br>Extended indicator support will be enabled.<br>**SQL_FALSE**<br>Extended indicator support is not enabled. SQL_FALSE is the default value. |
| 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.<br>This can be used as a governor for the maximum number of connections on a per application basis.<br>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.<br>**IBM specific:** This attribute is an IBM-defined extension.<br>**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. |

*Table 214. Connection attributes (continued)*

| ***Attribute*** | ***ValuePtr*** |
|---|---|
| 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:<br>**SQL_ONEPHASE**<br>The Db2 ODBC 3.0 driver does not support SQL_ONEPHASE.<br>**SQL_TWOPHASE**<br>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 used 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.<br>This attribute must be set before a connect request. Otherwise the attribute set request is rejected.<br>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.<br>**Recommendation:** Ensure that your application sets the SQL_ATTR_CONNECTTYPE attribute at the environment level rather than at a connection level. |

*Table 214. 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:<br>**SQL_TXN_READ_UNCOMMITTED**<br>Dirty reads, reads that cannot be repeated, and phantoms are possible.<br>**SQL_TXN_READ_COMMITTED**<br>Dirty reads are not possible. Reads that cannot be repeated, and phantoms are possible.<br>This is the default.<br>**SQL_TXN_REPEATABLE_READ**<br>Dirty reads and reads that cannot be repeated are not possible. Phantoms are possible.<br>**SQL_TXN_SERIALIZABLE**<br>Transactions can be serialized. Dirty reads, non-repeatable reads, and phantoms are not possible.<br>**SQL_TXN_NOCOMMIT**<br>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 ANSI/ISO SQL standard of 1992 isolation level, but an IBM defined extension, supported only by DB2 for i.<br>In IBM terminology,<br>• SQL_TXN_READ_UNCOMMITTED is "uncommitted read."<br>• SQL_TXN_READ_COMMITTED is "cursor stability."<br>• SQL_TXN_REPEATABLE_READ is "read stability."<br>• SQL_TXN_SERIALIZABLE is "repeatable read."<br>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).<br>**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()`. |

*Table 214. Connection attributes (continued)*

| *Attribute* | *ValuePtr* |
|---|---|

**Notes:**

1. You can change the default value for this attribute with the AUTOCOMMIT keyword in the ODBC initialization file.
2. You can change the default value for this attribute with the CONNECTTYPE keyword in the ODBC initialization file.
3. You can change the default value for this attribute with the MAXCONN keyword in the ODBC initialization file.
4. You can change the default value for this attribute with the TXNISOLATION keyword in the ODBC initialization file.

#### Return codes

After you call `SQLSetConnectAttr()`, it returns one of the following values:

- SQL_SUCCESS
- SQL_INVALID_HANDLE
- SQL_ERROR

#### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 215. `SQLSetConnectAttr()` SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (`SQLSetConnectAttr()` returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **01**S02 | Option value changed. | SQL_ATTR_SYNC_POINT changed to SQL_TWOPHASE. SQL_ONEPHASE is not supported. |
| **08**S01 | Unable to connect to data source. | The communication link between the application and the data source failed before the function completed. |
| **08**003 | Connection is closed. | An *Attribute* value is specified that requires an open connection, but the *ConnectionHandle* is not in a connected state. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate memory for the specified handle. |
| **HY**009 | Invalid use of a null pointer. | A null pointer is passed for *ValuePtr* and the value in **ValuePtr* is a string value. |
| **HY**010 | 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. |
| **HY**011 | Operation invalid at this time. | The argument *Attribute* is SQL_ATTR_TXN_ISOLATION and a transaction is open. |
| **HY**024 | Invalid attribute value. | Given the specified *Attribute* value, an invalid value is specified in **ValuePtr*. |

*Table 215. SQLSetConnectAttr() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**090 | Invalid string or buffer length. | The *StringLength* argument is less than 0, but is not SQL_NTS. |
| **HY**092 | Option type out of range. | The value specified for the argument *Attribute* is not valid for this version of Db2 ODBC. |
| **HY**C00 | 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. |

**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 concepts**

Extended indicators in ODBC applications
ODBC applications can use extended indicators to update all columns in UPDATE, INSERT, and MERGE statements without specifying the current value of columns that do not require changes.

Disable autocommit to reduce network flow
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 isolation levels for maximum concurrency and data consistency
Isolation levels determine the level of locking that is required to execute a statement and the level of concurrency that is possible in your application. You need to choose isolation levels for your application that maximize concurrency and that also ensure data consistency.

Distributed unit of work (Introduction to Db2 for z/OS)

**Related tasks**

Improving concurrency for applications that tolerate incomplete results (Db2 Performance)

Accessing currently committed data to avoid lock contention (Db2 Performance)

**Related reference**

Changes to SQLSetConnectAttr() attributes
For `SQLSetConnectAttr()` attributes, the ODBC driver supports both ODBC 2.0 and ODBC 3.0 values.

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

SQLGetConnectAttr() - Get current attribute setting
`SQLGetConnectAttr()` returns the current setting of a connection attribute and also allows you to set these attributes.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetStmtAttr() - Set statement attributes
`SQLSetStmtAttr()` sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call `SQLSetConnectAttr()`.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

isolation-clause (Db2 SQL)

# SQLSetConnection() - Set connection handle

SQLSetConnection() 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.

### ODBC specifications for SQLSetConnection()

*Table 216. SQLSetConnection() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| No | No | No |

### Syntax

```
SQLRETURN  SQLSetConnection  (SQLHDBC          hdbc);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 217. SQLSetConnection() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHDBC | *hdbc* | input | The connection handle associated with the connection to which the application wants 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.

### Return codes

After you call SQLSetConnection(), it returns one of the following values:

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 218. SQLSetConnection() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**003 | Connection is closed. | The connection handle provided is not currently associated with an open connection to a database server. |
| **HY**000 | 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. |

**Example**

The topic *Using arrays to pass parameter values* contains an example that demonstrates how to invoke SQLSetConnection().

**Related concepts**

Using arrays to pass parameter values
Db2 ODBC provides an array input method for updating Db2 tables.

Embedded SQL and Db2 ODBC in the same program
You can combine embedded static SQL with Db2 ODBC to write a mixed application.For 64-bit applications, you cannot use embedded static SQL statements.

**Related reference**

SQLConnect() - Connect to a data source
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 can call SQLConnect(). Subsequently, you must call SQLConnect() before you allocate a statement handle.

SQLDriverConnect() - Use a connection string to connect to a data source
SQLDriverConnect() is an alternative to SQLConnect(). Both functions establish a connection to the target database, but SQLDriverConnect() supports additional connection parameters.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLSetConnectOption() - Set connection option

This function is deprecated and is replaced by SQLSetConnectAttr(). You cannot use SQLSetConnectOption() for 64-bit applications.

**ODBC specifications for SQLSetConnectOption()**

*Table 219. SQLSetConnectOption() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | No |

**Syntax**

```
SQLRETURN   SQLSetConnectOption(
                              SQLHDBC           hdbc,
                              SQLUSMALLINT      fOption,
                              SQLUINTEGER       vParam);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 220. 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 null-terminated string. |

### Related reference

SQLSetConnectAttr() - Set connection attributes
`SQLSetConnectAttr()` sets attributes that govern aspects of connections.

## SQLSetCursorName() - Set cursor name

`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.

### ODBC specifications for SQLSetCursorName()

*Table 221. SQLSetCursorName() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLSetCursorName (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCursor,
                              SQLSMALLINT       cbCursor);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 222. 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 223. SQLSetCursorName() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **34**000 | Invalid cursor name. | This SQLSTATE is returned for one or more of the following reasons:<br>• 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.<br>• The cursor name specified by the argument *szCursor* already exists.<br>• The cursor name length is greater than the value returned by SQLGetInfo() with the SQL_MAX_CURSOR_NAME_LEN argument. |

*Table 223. SQLSetCursorName() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **58**004 | Unexpected system failure. | Unrecoverable system error. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**009 | Invalid use of a null pointer. | *szCursor* is a null pointer. |
| **HY**010 | Function sequence error. | This SQLSTATE is returned for one or more of the following reasons:<br>• There is an open or positioned cursor on the statement handle.<br>• 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.) |
| **HY**013 | Unexpected memory handling error. | Db2 ODBC is not able to access the memory that is required to support execution or completion of the function. |
| **HY**090 | Invalid string or buffer length. | The argument *cbCursor* is less than **0**, but not equal to SQL_NTS. |

**Example**

The following example 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 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:
        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 32. An application that sets a cursor name*

**Related reference**

SQLGetCursorName() - Get cursor name

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.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

# SQLSetEnvAttr() - Set environment attributes

SQLSetEnvAttr() sets attributes that affects all connections in an environment.

### ODBC specifications for SQLSetEnvAttr()

*Table 224. SQLSetEnvAttr() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| No | Yes | Yes |

### Syntax

```
SQLRETURN  SQLSetEnvAttr (SQLHENV           EnvironmentHandle,
                          SQLINTEGER        Attribute,
                          SQLPOINTER        ValuePtr,
                          SQLINTEGER        StringLength);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 225. SQLSetEnvAttr() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHENV | *EnvironmentHandle* | input | Environment handle. |
| SQLINTEGER | *Attribute* | input | Environment attribute to set. See Table 226 on page 368 for the list of attributes and their descriptions. |
| SQLPOINTER | *ValuePtr* | input | The 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 attribute value affects all connections in this environment.

The application can obtain the current attribute value by calling SQLGetEnvAttr().

Table 226 on page 368 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, refer to "Changes to SQLSetEnvAttr() attributes".

*Table 226. 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.<br><br>The following values are used to set the value of this attribute:<br><br>• SQL_OV_ODBC3: Causes the following ODBC 3.0 behavior:<br>– Db2 ODBC returns and expects ODBC 3.0 data type codes for date, time, and timestamp.<br>– Db2 ODBC returns ODBC 3.0 SQLSTATE codes when `SQLGetDiagRec()` is called.<br>– The *CatalogName* argument in a call to `SQLTables()` accepts a search pattern.<br>• SQL_OV_ODBC2 causes the following ODBC **2.x** behavior:<br>– Db2 ODBC returns and expects ODBC **2.x** data type codes for date, time, and timestamp.<br>– Db2 ODBC returns ODBC 2.0 SQLSTATE codes when `SQLGetDiagRec()` or `SQLError()` are called.<br>– 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:<br><br>• **SQL_TRUE**: Db2 ODBC uses nul-termination to indicate the length of output character strings.<br><br>This is the default.<br><br>• SQL_FALSE: Db2 ODBC does not use nul-termination in output character strings.<br><br>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.<br><br>This attribute can only be set when no connection handles are allocated under the environment handle. |

*Table 226. 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:<br>• **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*.<br>This is the default.<br>• SQL_COORDINATED_TRANS: The application wants 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.<br>This attribute must be set before allocating any connection handles, otherwise, the `SQLSetEnvAttr()` call is rejected.<br>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.<br>**IBM specific:** This attribute is an IBM-defined extension.<br>**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.<br>This can be used as a governor for the maximum number of connections on a per application basis.<br>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.<br>**IBM specific:** This attribute is an IBM-defined extension.<br>**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. |

*Table 226. Environment attributes (continued)*

| *Attribute* | Contents |
|---|---|

**Notes:**

1. You can change the default value for this attribute with the CONNECTTYPE keyword in the ODBC initialization file.
2. You can change the default value for this attribute with the MAXCONN keyword in the ODBC initialization file.

### Return codes

After you call SQLSetEnvAttr(), it returns one of the following values:

- SQL_SUCCESS
- SQL_INVALID_HANDLE
- SQL_ERROR

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 227. SQLSetEnvAttr() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**009 | Invalid use of a null pointer. | A null pointer is passed for *ValuePtr* and the value in **ValuePtr* is a string value. |
| **HY**011 | Operation invalid at this time. | Applications cannot set environment attributes while connection handles are allocated on the environment handle. |
| **HY**024 | Invalid attribute value. | Given the specified *Attribute* value, an invalid value is specified in **ValuePtr*. |
| **HY**090 | Invalid string or buffer length. | The *StringLength* argument is less than 0, but is not SQL_NTS. |
| **HY**092 | Option type out of range. | The value specified for the argument *Attribute* is not valid for this version of Db2 ODBC. |
| **HY**C00 | 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. |

### Example

The following example uses SQLSetEnvAttr() to set an environment attribute. Also, see the topic *Functions for establishing a distributed unit-of-work connection*.

```
SQLINTEGER output_nts,autocommit;
rc = SQLSetEnvAttr( henv,
                    SQL_ATTR_OUTPUT_NTS,
                    ( SQLPOINTER ) output_nts,
                    0
                   ) ;
CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc );
```

### Related concepts

Functions for 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.

Db2 ODBC restrictions on the ODBC connection model
Db2 ODBC does not fully support the ODBC connection model if the initialization file does not specify MULTICONTEXT=1.

**Related reference**

Changes to SQLSetEnvAttr() attributes
For SQLSetEnvAttr() attributes, the ODBC driver supports both ODBC 2.0 and ODBC 3.0 values.

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

SQLGetEnvAttr() - Return current setting of an environment attribute
SQLGetEnvAttr() returns the current setting for an environment attribute. You can also use the SQLSetEnvAttr() function to set these attributes.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetStmtAttr() - Set statement attributes
SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

# SQLSetParam() - Bind a parameter marker to a buffer

SQLSetParam() is a deprecated function and is replaced with SQLBindParameter().

### ODBC specifications for SQLSetParam()

*Table 228. SQLSetParam() specifications*

| ODBC | X/OPEN CLI | ISO CLI |
|---|---|---|
| 1.0 (Deprecated) | Yes | No |

### Example

Suppose that a program contains the following statement:

```
rc = SQLSetParam(hstmt, 1, SQL_C_LONG, SQL_INTEGER,
  0, 0, Prod_Num, NULL);
```

To get the same result with SQLBindParameter(), use this code:

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

### Related reference

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

SQLBindParameter() binds parameter markers to application variables and extends the capability of the SQLSetParam() function.

## SQLSetPos - Set the cursor position in a rowset

SQLSetPos() sets the cursor position in a rowset. Once the cursor is set, the application can refresh, update, and delete data in the rows.

### ODBC specifications for SQLSetPos()

*Table 229. SQLSetPos() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | Yes |

### Syntax

```
SQLRETURN   SQLSetPos (
               SQLHSTMT          StatementHandle,
               SQLSETPOSIROW     RowNumber,
               SQLUSMALLINT      Operation,
               SQLUSMALLINT      LockType);
```

### Function arguments

*Table 230. SQLSetPos arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *StatementHandle* | input | Statement handle. |
| SQLUSMALLINT | *RowNumber* | input | Position in the rowset of the row on which the operation that is specified by *Operation* is performed. If *RowNumber* is 0, the operation applies to every row in the rowset. |
| SQLUSMALLINT | *Operation* | input | Operation to perform:<br>• SQL_POSITION<br>• SQL_REFRESH<br>• SQL_UPDATE<br>• SQL_DELETE<br>• SQL_ADD<br>ODBC also specifies the following operations for backwards compatibility only, which Db2 ODBC also supports:<br>• SQL_ADD<br>Although Db2 ODBC supports SQL_ADD in SQLSetPos() calls, this function is deprecated. Use SQLBulkOperations() with an *Operation* value of SQL_ADD instead. |

*Table 230. SQLSetPos arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLUSMALLINT | *LockType* | input | Specifies how to lock the row after performing the operation that is specified in the *Operation* argument. Db2 ODBC supports SQL_LOCK_NO_CHANGE.<br>ODBC also specifies the following operations, which Db2 ODBC does not support:<br>• SQL_LOCK_EXCLUSIVE<br>• SQL_LOCK_UNLOCK |

**Usage**

***RowNumber* argument:** The *RowNumber* argument specifies the number of the row in the rowset on which to perform the operation that is specified by the *Operation* argument. If *RowNumber* is 0, the operation applies to every row in the rowset. *RowNumber* must be a value between 0 and the number of rows in the rowset, inclusive.

In the C language, arrays are 0-based, but the *RowNumber* argument 1-based. For example, to update the fifth row of the rowset, an application modifies the rowset buffers at array index 4, but specifies a *RowNumber* of 5.

An application can specify a cursor position when it calls `SQLSetPos()`. Generally, the application calls `SQLSetPos()` with the SQL_POSITION or SQL_REFRESH operation to position the cursor before executing a positioned update or delete statement or calling `SQLGetData()`.

***Operation* argument:** To determine which options are supported by a data source, an application calls `SQLGetInfo()` with one of the following information types, depending on the type of cursor:

- SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1
- SQL_DYNAMIC_CURSOR_ATTRIBUTES1
- SQL_STATIC_CURSOR_ATTRIBUTES1

*Operation* can have one of the following values:

**SQL_POSITION**

Db2 ODBC positions the cursor on the row specified by *RowNumber*.

The contents of the row status array that is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute are unchanged. The contents of the row operation array that is pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute are ignored.

**SQL_REFRESH**

Db2 ODBC positions the cursor on the row that is specified by *RowNumber*, and refreshes data in the rowset buffers for that row. For more information about how Db2 ODBC returns data in the rowset buffers, see the descriptions of row-wise and column-wise binding. Db2 ODBC supports only static cursors on SQL_REFRESH.

`SQLSetPos()` with an *Operation* value of SQL_REFRESH updates the status and content of the rows within the current fetched rowset. The data in the buffers is refreshed, *but not refetched*, so the membership in the rowset is fixed.

The contents of the row status array that is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute are also refreshed.

**SQL_UPDATE**

Db2 ODBC positions the cursor on the row that is specified by *RowNumber*, and updates the underlying row of data with the values in the rowset buffers (the *rgbValue* argument in

SQLBindCol()). SQLSetPos() retrieves the lengths of the data from the length or indicator buffers (the *pcbValue* argument in SQLBindCol()). If the length of any column is SQL_COLUMN_IGNORE, the column is not updated.

After the row is updated, the corresponding element of the row status array is updated to SQL_ROW_UPDATED or SQL_ROW_SUCCESS_WITH_INFO, if the row status array exists.

The row operation array that is pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute can be used to indicate that a row in the current rowset should be ignored during a bulk update.

**SQL_DELETE**

Db2 ODBC positions the cursor on the row that is specified by *RowNumber*, and deletes the underlying row of data.

The corresponding element of the row status array, which is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute, is changed to SQL_ROW_DELETED.

The row operation array that is pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute can be used to indicate that a row in the current rowset should be ignored during a bulk delete.

**SQL_ADD**

Db2 ODBC specifies the SQL_ADD value for backward compatibility only.

**Recommendation:** Instead of calling SQLPos() with the SQL_ADD value, use the ODBC 3.0 function SQLBulkOperations(), with the *Operation* argument set to SQL_ADD.

***LockType* argument**

The *LockType* argument provides a way for applications to control concurrency. Generally, data sources that support concurrency levels and transactions support only the SQL_LOCK_NO_CHANGE value of the *LockType* argument.

Although the *LockType* argument is specified for a single statement, the lock accords the same privileges to all statements on the connection. In particular, a lock that is acquired by one statement on a connection can be unlocked by a different statement on the same connection.

ODBC defines the following *LockType* arguments. Db2 ODBC supports only SQL_LOCK_NO_CHANGE. To determine which locks are supported by a data source, an application calls SQLGetInfo() with the SQL_LOCK_TYPES information type.

*Table 231. Operation values*

| LockType argument | Lock type |
|---|---|
| SQL_LOCK_NO_CHANGE | Ensures that the row is in the same locked or unlocked state as it was before SQLSetPos() was called. This value of *LockType* allows data sources that do not support explicit row-level locking to use the locking that is required by the current concurrency and transaction isolation levels. |
| SQL_LOCK_EXCLUSIVE | Not supported by Db2 ODBC. Locks the row exclusively. |
| SQL_LOCK_UNLOCK | Not supported by Db2 ODBC. Unlocks the row. |

**Status and operation arrays**

The following status and operation arrays are used with SQLSetPos():

- The row status array contains status values for each row of data in the rowset. Status values are set in this array after a call to SQLFetch(), SQLFetchScroll(), or SQLSetPos().
- The row operation array contains a value for each row in the rowset, which indicates whether a call to SQLSetPos() for a bulk operation is ignored or performed. Each element in the array is

SQL_ROW_PROCEED (the default) or SQL_ROW_IGNORE. The SQL_ATTR_ROW_OPERATION_PTR statement attribute points to the row operation array.

The number of elements in the status and operation arrays should be equal to the number of rows in the rowset, as defined by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.

### Return codes

After you call SQLPos(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 232. SQLSetPos SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (Function returns SQL_SUCCESS_WITH_INFO.) |
| **01**004 | Data truncated. | The *Operation* argument was SQL_REFRESH, and string or binary data that was returned for a column or columns with a data type of SQL_C_CHAR or SQL_C_BINARY resulted in the truncation of non-blank character or non-NULL binary data. |
| **01**S01 | Error in row. | The *RowNumber* argument was 0, and an error occurred in one or more rows when the operation that is specified with the *Operation* argument was performed.<br><br>SQL_SUCCESS_WITH_INFO is returned if an error occurs at least one, but not all, rows of a multirow operation. SQL_ERROR is returned if an error occurs on a single-row operation. |
| **01**S07 | Fractional truncation. | The *Operation* argument was SQL_REFRESH, the data type of the application buffer was not SQL_C_CHAR or SQL_C_BINARY, and the data that was returned to application buffers for one or more columns was truncated. For numeric data types, the fractional part of the number was truncated. For time and timestamp data types, the fractional portion of the time was truncated. |
| **07**006 | Invalid conversion. | The data value of a column in the result set could not be converted to the data type that was specified by *fCType* in the call to SQLBindCol(). |
| **21**S02 | Degrees of derived table does not match column list. | The argument *Operation* was SQL_UPDATE, and no columns could be updated because all columns were either unbound, read-only, or the value in the bound length or indicator buffer was SQL_COLUMN_IGNORE. |

*Table 232. SQLSetPos SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **22**001 | String data right truncation. | The assignment of a character or binary value to a column resulted in the truncation of non-blank (for characters) or non-null (for binary) characters or bytes. |
| **22**003 | Numeric value out of range. | One of the following conditions occurred:<br>• The argument *Operation* was SQL_UPDATE. The assignment of a numeric value to a column in the result set caused the whole (as opposed to fractional) part of the number to be truncated.<br>• The argument *Operation* was SQL_REFRESH. The numeric value for one or more bound columns could not be returned because significant digits were lost. |
| **22**008 | Invalid datetime format or datetime field overflow. | One of the following conditions occurred:<br>• The *Operation* argument was SQL_UPDATE. A datetime arithmetic operation on data that was sent to a column in the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result that was outside the permissible range of values for the field, or was invalid based on the natural rules for datetime values for the Gregorian calendar. Alternatively, the assignment of a numeric value to a column in the result set caused the whole part of the number to be truncated.<br>• The *Operation* argument was SQL_REFRESH. A datetime arithmetic operation on data that was retrieved from the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result that was outside the permissible range of values for the field, or was invalid based on the natural rules for datetime values for the Gregorian calendar. |
| **HY**000 | General error. | An error occurred for which there was no specific SQLSTATE. The error message that was returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause. |
| **HY**001 | Memory allocation failure. | Db2 ODBC was unable to allocate memory required to support execution or completion of the function. Process-level memory might have been exhausted for the application process. Consult the operating system configuration for information on process-level memory limitations. |

*Table 232. SQLSetPos SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**010 | Function sequence error. | One of the following conditions occurred:<br>• The specified *StatementHandle* was not in an executed state. The function was called without a previous call of SQLExecDirect(), SQLExecute(), or a catalog function.<br>• SQLExecute(), SQLExecDirect(), or SQLSetPos() was called for the *StatementHandle*, and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.<br>• SQLSetPos() was called for a *StatementHandle* before SQLFetchScroll() was called, or after SQLFetch() was called, and before SQLFreeStmt() was called with the SQL_CLOSE option. |
| **HY**011 | Operation invalid at this time. | The application set the SQL_ATTR_ROW_STATUS_PTR statement attribute. Then SQLSetPos() was called before SQLFetch(), SQLFetchScroll(), or SQLExtendedFetch() was called. |
| **HY**090 | Invalid string or buffer length. | One of the following conditions occurred:<br>• The *Operation* argument was SQL_ADD or SQL_UPDATE, a data value was a null pointer, and the column length value was not 0, SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.<br>• The *Operation* argument was SQL_ADD or SQL_UPDATE, a data value was not a null pointer, and the column length value was less than 0, but not equal to SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NTS, or SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. |
| **HY**107 | Row value out of range. | One of the following conditions occurred:<br>• The cursor that was associated with the *StatementHandle* was defined as forward only, so the cursor could not be positioned within the rowset. See the description for the SQL_ATTR_CURSOR_TYPE attribute in SQLSetStmtAttr().<br>• The *Operation* argument was SQL_UPDATE, SQL_DELETE, or SQL_REFRESH, and the row that was identified by the *RowNumber* argument was deleted or had not been fetched.<br>• The *Operation* argument was SQL_POSITION, and the *RowNumber* argument was 0. |
| **HY**C00 | Driver not capable. | Db2 ODBC or the data source does not support the operation that was requested in the *Operation* argument or the *LockType* argument. |

**Restrictions**

SQL_REFRESH for dynamic scrollable cursors is not supported by Db2 ODBC.

### Example

```
rc = SQLSetPos(
  hstmt,
  1,                  /* Position at the first row of the rowset. */
  SQL_POSITION,
  SQL_LOCK_NO_CHANGE); /* Do not change the lock state. */
```

### Related concepts

The ODBC row status array
The row status array returns the status of each row in the rowset.

### Related tasks

Providing long data for bulk inserts and positioned updates
To provide long data for bulk inserts or positioned updates, use SQLBulkOperations() or SQLSetPos() calls.

### Related reference

SQLBindCol() - Bind a column to an application variable
SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLCancel() - Cancel statement
SQLCancel() terminates an SQLExecDirect() or SQLExecute() sequence prematurely.

SQLDescribeCol() - Describe column attributes
SQLDescribeCol() returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either SQLPrepare() or SQLExecDirect().

SQLExecDirect() - Execute a statement directly
SQLExecDirect() prepares and executes an SQL statement in one step.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLNumResultCols() - Get number of result columns
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 you call SQLNumResultCols(). After you call SQLNumResultCols(), you can call SQLColAttribute() or one of the bind column functions.

## SQLSetStmtAttr() - Set statement attributes

SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

### ODBC specifications for SQLSetStmtAttr()

*Table 233. SQLSetStmtAttr() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 3.0 | Yes | Yes |

### Syntax

```
SQLRETURN  SQLSetStmtAttr (SQLHSTMT          StatementHandle,
                           SQLINTEGER        Attribute,
                           SQLPOINTER        ValuePtr,
                           SQLINTEGER        StringLength);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 234. SQLSetStmtAttr() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *StatementHandle* | input | Statement handle. |
| SQLINTEGER | *Attribute* | input | Statement attribute to set. Refer to Table 235 on page 380 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.<br>• For ODBC-defined attributes:<br>– If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*.<br>– If *ValuePtr* points to an integer, *BufferLength* is ignored.<br>• For driver-defined attributes (IBM extension):<br>– 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.<br>– 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 attribute descriptions shown in Table 235 on page 380. 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, refer to "Changes to `SQLSetStmtAttr()` attributes".

**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, using the statement attributes SQL_CCSID_CHAR and SQL_CCSID_GRAPHIC. 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()` or `SQLBindFileToParam()`, output data that is bound to columns through `SQLBindCol()` or `SQLBindFileToCol()`, and output data that is retrieved through `SQLGetData()`.

The CCSID override applies on a statement level only. Db2 will continue to use the default CCSID settings in the Db2 DSNHDECP 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.

The following table lists each *Attribute* value `SQLSetStmtAttr()` can set. Values shown in **bold** are default values.

*Table 235. 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.<br><br>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 incremented 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.) |
| SQL_CCSID_CHAR | A 32-bit integer value that specifies the CCSID of:<br><br>• Data that is bound to parameter markers with `SQLBindParameter()` or `SQLBindFileToParam()`<br>• Data that is bound to columns of a result set with `SQLBindCol()` or `SQLBindFileToCol()`<br>• Data that is retrieved with `SQLGetData()`<br><br>The CCSID applies to data for which the C symbolic data type is SQL_C_CHAR and the SQL data type is one of the following types:<br><br>• SQL_CHAR<br>• SQL_VARCHAR<br>• SQL_LONGVARCHAR<br>• SQL_CLOB<br>• SQL_TYPE_DATE<br>• SQL_TYPE_TIME<br>• SQL_TYPE_TIMESTAMP |

*Table 235. Statement attributes (continued)*

| Attribute | ValuePtr contents |
|---|---|
| SQL_CCSID_GRAPHIC | A 32-bit integer value that specifies the CCSID of:<br>• Data that is bound to parameter markers with `SQLBindParameter()` or `SQLBindFileToParam()`<br>• Data that is bound to columns of a result set with `SQLBindCol()` or `SQLBindFileToCol()`<br>• Data that is retrieved with `SQLGetData()`<br>The CCSID applies to data for which the C symbolic data type is SQL_C_DBCHAR and the SQL data type is one of the following types:<br>• SQL_GRAPHIC<br>• SQL_VARGRAPHIC<br>• SQL_LONGVARGRAPHIC<br>• SQL_DBCLOB<br>• SQL_TYPE_DATE<br>• SQL_TYPE_TIME<br>• SQL_TYPE_TIMESTAMP |
| SQL_ATTR_CLOSE_BEHAVIOR | A 32-bit integer value that forces the release of locks upon an underlying CLOSE CURSOR operation. The possible values are:<br>• **SQL_CC_NO_RELEASE:** locks are not released when the cursor on this statement handle is closed.<br>• SQL_CC_RELEASE: locks are released when the cursor on this statement handle is closed.<br>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 235. Statement attributes (continued)*

| Attribute | ValuePtr contents |
|---|---|
| SQL_ATTR_CONCURRENCY | A 32-bit integer value that specifies the cursor concurrency:<br>• **SQL_CONCUR_READ_ONLY** - Cursor is read-only. No updates are allowed. Supported for forward-only and static cursors.<br>• SQL_CONCUR_LOCK - Cursor uses the lowest level of locking sufficient to ensure that the row can be updated. Supported for forward-only and dynamic cursors.<br>The default value for SQL_ATTR_CONCURRENCY is SQL_CONCUR_READ_ONLY for static and forward-only cursors. The default for dynamic cursors is SQL_CONCUR_LOCK.<br>If the SQL_ATTR_CURSOR_TYPE attribute is changed to a type that does not support the current value of SQL_ATTR_CONCURRENCY, the value of SQL_ATTR_CONCURRENCY is changed at execution time, and a warning is issued when `SQLExecDirect()` or `SQLPrepare()` is called.<br>If a SELECT FOR UPDATE statement is executed when the value of SQL_ATTR_CONCURRENCY is set to SQL_CONCUR_READ_ONLY, an error is returned.<br>If the value of SQL_ATTR_CONCURRENCY is changed to a value that is supported for some value of SQL_ATTR_CURSOR_TYPE, but not for the current value of SQL_ATTR_CURSOR_TYPE, the value of SQL_ATTR_CURSOR_TYPE is changed at execution time, and a warning is issued when `SQLExecDirect()` or `SQLPrepare()` is called.<br>If the specified concurrency is not supported by the data source, Db2 ODBC substitutes a different concurrency and returns a warning. The order of substitution depends on the cursor type:<br>• Forward-only: SQL_CONCUR_LOCK is substituted for SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES.<br>• Static: SQL_CONCUR_READ_ONLY is substituted for SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES.<br>• Dynamic: SQL_CONCUR_LOCK is substituted for SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES.<br>**Unsupported attribute values:** ODBC architecture defines the following values, which are not supported by Db2 ODBC:<br>• SQL_CONCUR_VALUES - Cursor uses optimistic concurrency control, comparing values.<br>• SQL_CONCUR_ROWVER - Cursor uses optimistic concurrency control.<br>If one of these values is used, SQL_SUCCESS_WITH_INFO (SQLSTATE **01**S02) is returned and the option value is changed. |

*Table 235. Statement attributes (continued)*

| Attribute | ValuePtr contents |
|---|---|
| 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.<br>• **SQL_CURSOR_HOLD_ON**<br>• SQL_CURSOR_HOLD_OFF<br>The default value when a statement handle is first allocated is SQL_CURSOR_HOLD_ON.<br>This attribute cannot be specified while there is an open cursor on this statement handle. |
| SQL_ATTR_CURSOR_SCROLLABLE | A 32-bit integer that specifies the level of support that the application requires. Setting this attribute affects subsequent calls to `SQLExecDirect()` and `SQLExecute()`. The supported values are:<br>• **SQL_NONSCROLLABLE** - Scrollable cursors are not required on the statement handle. If the application calls `SQLFetchScroll()` on this handle, the only valid value of *FetchOrientation* is SQL_FETCH_NEXT.<br>• SQL_SCROLLABLE - Scrollable cursors are required on the statement handle. When the application calls `SQLFetchScroll()`, it can specify any valid value of *FetchOrientation*, for cursor positioning in modes other than the sequential mode. |
| SQL_ATTR_CURSOR_SENSITIVITY | A 32-bit integer that specifies whether changes that are made by other cursors are visible to the cursors on the statement handle. Setting this attribute affects subsequent calls to `SQLExecDirect()` and `SQLExecute()`. The supported values are:<br>• **SQL_UNSPECIFIED** - The cursor type, and whether changes that are made by other cursors are visible to the cursors on the statement handle, are unspecified. Cursors on the statement handle can make visible none, some or all such changes.<br>• SQL_INSENSITIVE - All cursors on the statement handle show the result set without reflecting any changes that are made to it by any other cursor. Insensitive cursors are read-only. This attribute corresponds to a static cursor that has a concurrency that is read-only.<br>• SQL_SENSITIVE - Corresponds to a static cursor that has a read-only concurrency. |

*Table 235. Statement attributes (continued)*

| Attribute | ValuePtr contents |
|---|---|
| SQL_ATTR_CURSOR_TYPE | A 32-bit integer value that specifies the cursor type. The supported values are:<br>• **SQL_CURSOR_FORWARD_ONLY** - Cursor behaves as a forward only scrolling cursor.<br>• SQL_CURSOR_STATIC - The data in the result set is static.<br>• SQL_CURSOR_DYNAMIC - The cursor detects all changes in the result set.<br>These options cannot be set if there is an open cursor on the associated statement handle.<br>If the specified cursor type is not supported by the data source, Db2 ODBC substitutes a different cursor type and returns a warning. For a dynamic cursor, Db2 ODBC substitutes a different cursor type, in the following order: a static cursor or a forward-only cursor.<br>**Unsupported attribute values:** ODBC architecture defines the SQL_CURSOR_KEYSET_DRIVEN value, which is not supported by Db2 ODBC. If this value is specified, Db2 ODBC sets the statement attribute to SQL_CURSOR_STATIC or SQL_CURSOR_FORWARD_ONLY, and returns SQLSTATE **01**S02 (Option value changed). In this case the application needs to call `SQLGetStmtAttr()` to query the value that is set. |
| 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. |
| 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()`.<br>• **SQL_NODESCRIBE_OFF**<br>• SQL_NODESCRIBE_ON<br>This attribute cannot be specified while there is an open cursor on this statement handle.<br>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 want to incur the extra network traffic associated with the descriptor information needed by Db2 ODBC to provide client side processing.<br>**IBM specific:** This attribute is an IBM-defined extension. |

*Table 235. Statement attributes (continued)*

| Attribute | ValuePtr contents |
|---|---|
| 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:<br>• **SQL_NOSCAN_OFF** - SQL strings are scanned for escape clause sequences.<br>• SQL_NOSCAN_ON - SQL strings are not scanned for escape clauses. Everything is sent directly to the server for processing.<br>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_PARAMOPT_ATOMIC | A 32-bit integer value that determines whether the application uses atomic or non-atomic SQL for the underlying processing of multi-row insert operations. The attribute value takes effect after `SQLSetStmtAttr()` is used to specify multiple values for parameter markers for an SQL INSERT statement. Possible values are:<br>**SQL_ATOMIC_YES**<br>The application uses atomic SQL for the underlying processing of multi-row insert operations. This is the default.<br>Specification of SQL_ATOMIC_YES for a connection to a server that does not support multi-row inserts results in SQLSTATE 01S02 (option value changed). The attribute value is set to SQL_ATOMIC_NO.<br>**SQL_ATOMIC_NO**<br>The application uses non-atomic SQL for the underlying processing of multi-row insert operations. |
| SQL_ATTR_PARAMSET_SIZE | A 32-bit unsigned integer value that specifies the number of values for each parameter. If SQL_ATTR_PARAMSET_SIZE is greater than 1, the *rgbValue* argument in SQLBindParameter() points to an array of parameter values and *pcbValue* argument points to an array of lengths. The cardinality of each array is equal to the value of this field. |
| SQL_ATTR_PARAMS_PROCESSED_PTR | A 32-bit unsigned integer * field that points to a buffer in which to return the current row number. As each row of parameters is processed, this is set to the number of that row. No row number is returned if this is a null pointer.<br>If the call to SQLExecDirect() or SQLExecute() that fills in the buffer pointed to by this attribute does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined. |

*Table 235. Statement attributes (continued)*

| Attribute | ValuePtr contents |
|---|---|
| SQL_ATTR_RETRIEVE_DATA | A 32-bit integer value that indicates whether Db2 ODBC should retrieve data from the database when SQLFetchScroll() or SQLExtendedFetch() is called. The possible values are:<br>• **SQL_RD_ON**: SQLFetchScroll() or SQLExtendedFetch() retrieves data after it positions the cursor to the specified location.<br>• SQL_RD_OFF: SQLFetchScroll() or SQLExtendedFetch() does not retrieve data after it positions the cursor. By setting SQL_RETRIEVE_DATA to SQL_RD_OFF, an application can verify whether a row exists, or retrieve a bookmark for the row without incurring the overhead of retrieving rows.<br>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 that are returned by each call toSQLFetchScroll(). The default value is 1.<br>If the specified rowset size exceeds the maximum rowset size that is supported by the data source, Db2 ODBC substitutes the maximum supported value and returns SQLSTATE **01**S02 (Option value changed). |
| SQL_ATTR_ROW_NUMBER | A 32-bit integer value that is the number of the current row in the entire result set. If the number of the current row cannot be determined, or there is no current row, Db2 ODBC returns 0. This attribute can be retrieved by a call to SQLGetStmtAttr(), but not set by a call to SQLSetStmtAttr(). |
| SQL_ATTR_ROW_OPERATION_POINTER | A 16-bit unsigned integer * value that points to an array of UDWORD values that are used to ignore a row when SQLSetPos() is used to perform a bulk operation. Each value is set to SQL_ROW_PROCEED (for the row to be included in the bulk operation) or SQL_ROW_IGNORE (for the row to be excluded from the bulk operation). During calls to SQLBulkOperations(), rows cannot be ignored by using this array.<br>If SQL_ATTR_ROW_OPERATION_POINTER is set to a null pointer, Db2 ODBC does not return row status values. This attribute can be set at any time, but the new value is not used until the next time SQLFetchScroll() or SQLSetPos() is called. |
| SQL_ATTR_ROW_STATUS_PTR | A 16-bit unsigned integer * value that points to an array of UWORD values that contain row status values after a call to SQLFetch() or SQLFetchScroll(). The array has as many elements as there are rows in the rowset.<br>If SQL_ATTR_ROW_STATUS_PTR is set to a null pointer, Db2 ODBC does not return row status values. This attribute can be set at any time, but the new value is not used until the next time SQLFetch(), SQLFetchScroll(),or SQLSetPos() is called. |

*Table 235. Statement attributes (continued)*

| Attribute | ValuePtr contents |
|---|---|
| SQL_ATTR_ROWS_FETCHED_PTR | A 32-bit unsigned integer * value that points to a buffer that contains the number of rows that were fetched after a call to `SQLFetch()` or `SQLFetchScroll()`. The array has as many elements as there are rows in the rowset.<br><br>Db2 ODBC maps SQL_ATTR_ROWS_FETCHED_PTR to the *RowCountPtr* array in a call to `SQLExtendedFetch()`. |
| 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 that is 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 even when the cursor is open and becomes effective on the next `SQLExtendedFetch()` call.<br><br>**Recommendation:** Use `SQLFetchScroll()` rather than `SQLExtendedFetch()`. Use the statement attribute SQL_ATTR_ROW_ARRAY_SIZE rather than SQL_ATTR_ROWSET_SIZE to set the number of rows in the rowset. |
| 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. For the permitted values, refer to the function `SQLSetConnectOption()`<br><br>This attribute cannot be set if there is an open cursor on this statement handle (SQLSTATE **24**000).<br><br>**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.<br><br>For more information about setting this attribute, refer to *Isolation levels for maximum concurrency and data consistency*. |
| SQL_ATTR_USE_BOOKMARKS | A 32-bit integer value that specifies whether an application uses bookmarks with a cursor. The only supported attribute value is SQL_UB_OFF, which indicates that an application does not use bookmarks with a cursor.<br><br>**Unsupported attribute value:** ODBC architecture defines the SQL_UB_VARIABLE value, which is not supported by Db2 ODBC. SQL_UB_VARIABLE indicates that an application uses bookmarks with a cursor, and that the ODBC driver provides variable-length bookmarks, if they are supported. |

**Notes:**

1. You can change the default value for this attribute with the CURSORHOLD keyword in the ODBC initialization file.
2. You can change the default value for this attribute with the TXNISOLATION keyword in the ODBC initialization file.

**Return codes**

After you call `SQLSetStmtAttr()`, it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 236. SQLSetStmtAttr() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **01**000 | Warning. | Informational message. (SQLSetStmtAttr() returns SQL_SUCCESS_WITH_INFO for this SQLSTATE.) |
| **01**S02 | 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.) |
| **08**S01 | Unable to connect to data source. | The communication link between the application and the data source failed before the function completed. |
| **24**000 | Invalid cursor state. | The *Attribute* is SQL_ATTR_CONCURRENCY and the cursor is open. |
| **HY**000 | 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. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate memory for the specified handle. |
| **HY**009 | Invalid use of a null pointer. | A null pointer is passed for *ValuePtr* and the value in **ValuePtr* is a string value. |
| **HY**010 | 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. |
| **HY**011 | Operation invalid at this time. | The *Attribute* is SQL_ATTR_CONCURRENCY and the statement is prepared. |
| **HY**024 | Invalid attribute value. | Given the specified *Attribute* value, an invalid value is specified in **ValuePtr*. |
| **HY**090 | Invalid string or buffer length. | The *StringLength* argument is less than 0, but is not SQL_NTS. |
| **HY**092 | Option type out of range. | The value specified for the argument *Attribute* is not valid for this version of Db2 ODBC. |
| **HY**C00 | 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. |

**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 concepts**

Disable cursor hold behavior for more efficient resource use
Db2 ODBC can more efficiently use resources that are associated with statement handles if you disable cursor-hold behavior for statements that do not require it.

Limit the number of rows that an application can fetch
If a result set is too large, it might cause problems for the application. You can follow guidelines to reduce errors.

Set isolation levels for maximum concurrency and data consistency
Isolation levels determine the level of locking that is required to execute a statement and the level of concurrency that is possible in your application. You need to choose isolation levels for your application that maximize concurrency and that also ensure data consistency.

**Related reference**

Changes to SQLSetStmtAttr() attributes
For SQLSetStmtAttr() attributes, the ODBC driver supports both ODBC 2.0 and ODBC 3.0 values.

SQLCancel() - Cancel statement
SQLCancel() terminates an SQLExecDirect() or SQLExecute() sequence prematurely.

SQLGetConnectAttr() - Get current attribute setting
SQLGetConnectAttr() returns the current setting of a connection attribute and also allows you to set these attributes.

SQLGetStmtAttr() - Get current setting of a statement attribute
SQLGetStmtAttr() returns the current setting of a statement attribute. To set these statement attributes, use SQLSetStmtAttr().

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSetConnectAttr() - Set connection attributes
SQLSetConnectAttr() sets attributes that govern aspects of connections.

SQLSetConnectOption() - Set connection option
This function is deprecated and is replaced by SQLSetConnectAttr(). You cannot use SQLSetConnectOption() for 64-bit applications.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

# SQLSetStmtOption() - Set statement attribute

This function is deprecated and is replaced by SQLSetStmtAttr(). You cannot use SQLSetStmtOption() for 64-bit applications.

**ODBC specifications for SQLSetStmtOption()**

*Table 237. SQLSetStmtOption() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | No |

### Syntax

```
SQLRETURN   SQLSetStmtOption (SQLHSTMT          hstmt,
                              SQLUSMALLINT      fOption,
                              SQLUINTEGER       vParam);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 238. 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. |

### Related reference

SQLSetStmtAttr() - Set statement attributes
`SQLSetStmtAttr()` sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call `SQLSetConnectAttr()`.

## SQLSpecialColumns() - Get special (row identifier) columns

`SQLSpecialColumns()` returns unique row identifier information (primary key or unique index) for a table. The information is returned in an SQL result set. You can retrieve this result set with the same functions that process a result set that is generated by a query.

### ODBC specifications for SQLSpecialColumns()

*Table 239. SQLSpecialColumns() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | No |

### 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

The following table lists the data type, use, and description for each argument in this function.

*Table 240. SQLSpecialColumns() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHSTMT | *hstmt* | input | Statement handle. |

*Table 240. SQLSpecialColumns() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLUSMALLINT | *fColType* | input | Type of unique row identifier to return. Only the following type is supported:<br>• SQL_BEST_ROWID, which returns the optimal set of columns that can uniquely identify any row in the specified table.<br>**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*. |
| SQLUSMALLINT | *fScope* | input | Minimum required duration for which the unique row identifier is valid.<br>*fScope* must be one of the following:<br>• 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.<br>• 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.<br>• SQL_SCOPE_SESSION: The row identifier is guaranteed to be valid for the duration of the connection.<br>**Important:** This attribute is not supported by Db2 for z/OS.<br>The duration over which a row identifier value is guaranteed to be valid depends on the current transaction isolation level. |
| SQLUSMALLINT | *fNullable* | input | Determines whether to return special columns that can have a null value.<br>Must be one of the following:<br>• SQL_NO_NULLS - The row identifier column set returned cannot have any null values.<br>• 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 241 on page 392 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 ANSI/ISO SQL standard of 1992 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 database management system.

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. The following table lists these columns.

*Table 241. 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:<br>• SQL_SCOPE_CURROW<br>• SQL_SCOPE_TRANSACTION<br>• SQL_SCOPE_SESSION<br>See *fScope* in Table 240 on page 390 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. |
| 4 | TYPE_NAME | VARCHAR(128) NOT NULL | database management system character string represented of the name associated with DATA_TYPE column value. |

*Table 241. Columns returned by SQLSpecialColumns() (continued)*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 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.<br>For date, time, timestamp data types, this is the total number of bytes required to display the value when converted to character.<br>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. |
| 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. |
| 7 | DECIMAL_DIGITS | SMALLINT | The scale of the column. NULL is returned for data types where scale is not applicable. |
| 8 | PSEUDO_COLUMN | SMALLINT | Indicates whether the column is a pseudo-column. Db2 ODBC only returns:<br>• SQL_PC_NOT_PSEUDO<br>Db2 database management systems do not support pseudo columns. ODBC applications can receive the following values from other non-IBM relational database management system servers:<br>• SQL_PC_UNKNOWN<br>• 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

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 242. SQLSpecialColumns() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is opened on the statement handle. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |
| **HY**090 | Invalid string or buffer length. | This SQLSTATE is returned for one or more of the following reasons:<br>• The value of one of the length arguments is less than 0, but not equal to SQL_NTS.<br>• The value of one of the length arguments exceeded the maximum length supported by the database management system for that qualifier or name. |
| **HY**097 | Column type out of range. | An invalid *fColType* value is specified. |
| **HY**098 | Scope type out of range. | An invalid *fScope* value is specified. |
| **HY**099 | Nullable type out of range. | An invalid *fNullable* values is specified. |
| **HY**C00 | Driver not capable. | Db2 ODBC does not support *catalog* as a qualifier for table name. |

**Example**

The following example 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
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("   name.s, type_name.s);
        if (precision_ind != SQL_NULL_DATA) {
            printf(" (
        } else {
            printf("(\n");
        }
        if (scale_ind != SQL_NULL_DATA) {
            printf(",
        } else {
            printf(")\n");
        }
    }
/* ... */
```

*Figure 33. An application that prints the column set for a unique index of a table*

**Related reference**

C and SQL data types
Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

Length of SQL data types
The length of a column is the maximum number of bytes that are returned to the application when data is transferred to its default C data type.

Precision of SQL data types
The precision of a numeric column or parameter refers to the maximum number of digits that are used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter.

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.

SQLColumns() - Get column information
`SQLColumns()` returns a list of columns in the specified tables. The information is returned in an SQL result set, which can be retrieved by using the same functions that fetch a result set that a query generates.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLStatistics() - Get index and statistics information for a base table
`SQLStatistics()` retrieves index information for a specific table. It also returns the cardinality and the number of pages that are associated with the table and the indexes on the table. The information is returned in a result set. You can retrieve the result set with the same functions that process a result set that is generated by a query.

SQLTables() - Get table information

SQLTables() returns a list of table names and associated information that is stored in the system catalog of the connected data source. The list of table names is returned as a result set. You can retrieve this result set with the same functions that process a result set generated by a query.

isolation-clause (Db2 SQL)

## SQLStatistics() - Get index and statistics information for a base table

SQLStatistics() retrieves index information for a specific table. It also returns the cardinality and the number of pages that are associated with the table and the indexes on the table. The information is returned in a result set. You can retrieve the result set with the same functions that process a result set that is generated by a query.

### ODBC specifications for SQLStatistics()

*Table 243. SQLStatistics() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | No |

### 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

The following table lists the data type, use, and description for each argument in this function.

*Table 244. 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:<br>• SQL_INDEX_UNIQUE<br>Only unique indexes are returned.<br>• SQL_INDEX_ALL<br>All indexes are returned. |

*Table 244. 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:<br>• 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.<br>• 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 245 on page 397 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 ANSI/ISO SQL standard of 1992 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 database management system supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected database management system supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected database management system supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected database management system 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. The following table lists the columns in the result set SQLStatistics() currently returns.

*Table 245. Columns returned by SQLStatistics()*

| Column number | Column name | Data type | Description |
|---|---|---|---|
| 1 | TABLE_CAT | VARCHAR(128) | The is always null. |

*Table 245. 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:<br>• SQL_TRUE if the index allows duplicate values.<br>• SQL_FALSE if the index values must be unique.<br>• 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:<br>• SQL_TABLE_STAT - Indicates this row contains statistics information on the table itself.<br>• SQL_INDEX_CLUSTERED - Indicates this row contains information on an index, and the index type is a clustered index.<br>• SQL_INDEX_HASHED - Indicates this row contains information on an index, and the index type is a hashed index.<br>• 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 245. 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.<br>• If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of unique values in the index.<br>• A null value is returned if information is not available from the database management system. |
| 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.<br>• If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of pages used to store the indexes.<br>• A null value is returned if information is not available from the database management system. |
| 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 for z/OS ODBC, 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

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 246. SQLStatistics() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is opened on the statement handle. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |
| **HY**090 | Invalid string or buffer length. | This SQLSTATE is returned for one or more of the following reasons:<br>• The value of one of the name length arguments is less than 0, but not equal to SQL_NTS.<br>• 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(). |
| **HY**100 | Uniqueness option type out of range. | An invalid *fUnique* value is specified. |
| **HY**101 | Accuracy option type out of range. | An invalid *fAccuracy* value is specified. |
| **HY**C00 | Driver not capable. | Db2 ODBC does not support *catalog* as a qualifier for table name. |

**Example**

The following example 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
    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 =
       else
          printf("    Cardinality = (Unavailable)");
       if (pages_ind != SQL_NULL_DATA)
          printf(" Pages =
       else
          printf(" Pages = (Unavailable)\n");
    }
/* ... */
```

*Figure 34. An application that prints page and cardinality information about a table*

**Related reference**

SQLColumns() - Get column information
`SQLColumns()` returns a list of columns in the specified tables. The information is returned in an SQL result set, which can be retrieved by using the same functions that fetch a result set that a query generates.

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLSpecialColumns() - Get special (row identifier) columns
`SQLSpecialColumns()` returns unique row identifier information (primary key or unique index) for a table. The information is returned in an SQL result set. You can retrieve this result set with the same functions that process a result set that is generated by a query.

# SQLTablePrivileges() - Get table privileges

`SQLTablePrivileges()` returns a list of tables and associated privileges for each table. The information is returned in an SQL result set. You can retrieve this result set with the same functions that you use to process a result set that is generated by a query.

**ODBC specifications for SQLTablePrivileges()**

*Table 247. SQLTablePrivileges() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | No | No |

### Syntax

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

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 248. 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 patterns.

### 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 ANSI/ISO SQL standard of 1992 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 database management system supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected database management system supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected database management system supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected database management system 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. The following table lists the columns in the result set SQLTablePrivileges() currently returns.

*Table 249. 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:<br>• ALTER<br>• CONTROL<br>• DELETE<br>• INDEX<br>• INSERT<br>• REFERENCES<br>• SELECT<br>• UPDATE |
| 7 | IS_GRANTABLE | VARCHAR(3) | Indicates whether the grantee is permitted to grant the privilege to other users.<br>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

**Diagnostics**

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 250. SQLTablePrivileges() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is opened on the statement handle. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |
| **HY**090 | Invalid string or buffer length. | This SQLSTATE is returned for one or more of the following reasons:<br>• The value of one of the name length arguments is less than 0, but not equal to SQL_NTS.<br>• 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. |
| **HY**C00 | Driver not capable. | Db2 ODBC does not support *catalog* as a qualifier for table name. |

**Example**

The following example 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 privileges 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 35. An application that generates a result set containing privileges on tables*

**Related concepts**

Input arguments on catalog functions

Input arguments identify or constrain the amount of information that a catalog function returns.

**Related reference**

Function return codes
Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLTables() - Get table information
`SQLTables()` returns a list of table names and associated information that is stored in the system catalog of the connected data source. The list of table names is returned as a result set. You can retrieve this result set with the same functions that process a result set generated by a query.

# SQLTables() - Get table information

`SQLTables()` returns a list of table names and associated information that is stored in the system catalog of the connected data source. The list of table names is returned as a result set. You can retrieve this result set with the same functions that process a result set generated by a query.

### ODBC specifications for SQLTables()

*Table 251. SQLTables() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 | Yes | No |

### 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

The following table lists the data type, use, and description for each argument in this function.

*Table 252. 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.<br>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*. |

*Table 252. SQLTables() arguments (continued)*

| Data type | Argument | Use | Description |
|---|---|---|---|
| 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*. |
| SQLCHAR * | *szTableType* | input | Buffer that can contain a *value list* to qualify the result set by table type.<br>The value list is a list of uppercase comma-separated single quoted values for the table types of interest.<br>Valid table type identifiers can include: TABLE, VIEW, SYSTEM TABLE, ALIAS, SYNONYM, GLOBAL TEMPORARY TABLE, AUXILIARY TABLE, MATERIALIZED QUERY TABLE, or ACCEL-ONLY TABLE.<br>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.

**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.

If an application calls SQLTables() with null pointers for some or all of the *szSchemaName*, *szTableName*, and *szTableType* arguments, SQLTables() does not restrict the result set that is returned. For some data sources that contain a large number of objects, large result sets are returned, with very long retrieval times. You can reduce the result set size and retrieval time by specifying initialization keywords SCHEMALIST, SYSSCHEMA, or TABLETYPE in the Db2 ODBC initialization file. Those initialization keywords restrict the result set when SQLTables() supplies null pointers for *szSchemaName* and *szTableType*. If SQLTables() does not supply a null pointer for *szSchemaName* or *szTableType*, the associated keyword specification in the Db2 ODBC initialization file is not used.

The result set returned by SQLTables() contains the columns listed in Table 253 on page 407 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 ANSI/ISO SQL standard of 1992 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 database management system supports
- SQL_MAX_SCHEMA_NAME_LEN, to determine the length of TABLE_SCHEM columns that the connected database management system supports
- SQL_MAX_TABLE_NAME_LEN, to determine the length of TABLE_NAME columns that the connected database management system supports
- SQL_MAX_COLUMN_NAME_LEN, to determine the length of COLUMN_NAME columns that the connected database management system 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. The following table lists the columns in the result set `SQLTables()` currently returns.

*Table 253. 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 of object in the TABLE_NAME column. TABLE_TYPE can have one of the string values 'TABLE', 'VIEW', 'INOPERATIVE VIEW', 'SYSTEM TABLE', 'ALIAS', 'SYNONYM', 'GLOBAL TEMPORARY TABLE', 'AUXILIARY TABLE', 'MATERIALIZED QUERY TABLE', or 'ACCEL-ONLY TABLE'. 'ACCEL-ONLY TABLE' is an extended table type, and is returned only if initialization keyword EXTENDEDTABLEINFO is set to 1. |
| REMARKS | VARCHAR(762) | Contains the descriptive information about the table. |
| TEMPORAL_TABLE_TYPE | VARCHAR(11) | Contains the type of temporal table. Possible values are:<br>**SYSTEM**<br>System-period temporal table.<br>**APPLICATION**<br>Application-period temporal table.<br>**BITEMPORAL**<br>Bitemporal table.<br>**Empty string**<br>Not a temporal table.<br>The result set contains this column only if initialization keyword EXTENDEDTABLEINFO is set to 1. |

*Table 253. Columns returned by SQLTables() (continued)*

| Column Name | Data type | Description |
|---|---|---|
| IS_ACCELERATED | VARCHAR(3) | Indicates whether the table is an accelerated table. Possible values are YES or NO.<br>The result set contains this column only if initialization keyword EXTENDEDTABLEINFO is set to 1. |
| ACCEL_ARCHIVE_STATUS | CHAR(1) | Contains the archive status of the table in the accelerator database. See the description of the ARCHIVE column in SYSACCEL.SYSACCELERATEDTABLES table (Db2 SQL) for the possible values and their meanings.<br>The result set contains this column only if initialization keyword EXTENDEDTABLEINFO is set to 1. |
| IS_ARCHIVE_ENABLED | VARCHAR(3) | Indicates whether the table is an archive-enabled table. Possible values are YES or NO.<br>The result set contains this column only if initialization keyword EXTENDEDTABLEINFO is set to 1. |

### Return codes

After you call SQLTables(), it returns one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

### Diagnostics

The following table lists each SQLSTATE that this function generates, with a description and explanation for each value.

*Table 254. SQLTables() SQLSTATEs*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **08**S01 | Communication link failure. | The communication link between the application and data source fails before the function completes. |
| **24**000 | Invalid cursor state. | A cursor is open on the statement handle. |
| **HY**001 | Memory allocation failure. | Db2 ODBC is not able to allocate the required memory to support the execution or the completion of the function. |
| **HY**010 | 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.) |
| **HY**014 | No more handles. | Db2 ODBC is not able to allocate a handle due to low internal resources. |

*Table 254. SQLTables() SQLSTATEs (continued)*

| SQLSTATE | Description | Explanation |
|---|---|---|
| **HY**090 | Invalid string or buffer length. | This SQLSTATE is returned for one or more of the following reasons:<br>• The value of one of the name length arguments is less than 0, but not equal to SQL_NTS.<br>• 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(). |
| **HY**C00 | Driver not capable. | Db2 ODBC does not support *catalog* as a qualifier for table name. |

**Example**

The following example shows an application that uses SQLTables() to generate a result set of table name information that matches a search pattern. For another example, see *Functions for querying environment and data source information*.

```
/* ... */
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 36. An application that returns a result set of table name information*

**Related concepts**

Functions for querying environment and data source information
Db2 ODBC provides functions that let applications retrieve information about the characteristics and capabilities of the current ODBC driver or the data source to which it is connected.

Input arguments on catalog functions
Input arguments identify or constrain the amount of information that a catalog function returns.

**Related reference**

SQLColumns() - Get column information
SQLColumns() returns a list of columns in the specified tables. The information is returned in an SQL result set, which can be retrieved by using the same functions that fetch a result set that a query generates.

Function return codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

SQLTablePrivileges() - Get table privileges
SQLTablePrivileges() returns a list of tables and associated privileges for each table. The information is returned in an SQL result set. You can retrieve this result set with the same functions that you use to process a result set that is generated by a query.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

# SQLTransact() - Transaction management

SQLTransact() is a deprecated function and is replaced by SQLEndTran().

### ODBC specifications for SQLTransact()

*Table 255. SQLTransact() specifications*

| ODBC specification level | In X/Open CLI CAE specification? | In ISO CLI specification? |
|---|---|---|
| 1.0 (Deprecated) | Yes | Yes |

### Syntax

```
SQLRETURN   SQLTransact      (SQLHENV           henv,
                              SQLHDBC           hdbc,
                              SQLUSMALLINT      fType);
```

### Function arguments

The following table lists the data type, use, and description for each argument in this function.

*Table 256. SQLTransact() arguments*

| Data type | Argument | Use | Description |
|---|---|---|---|
| SQLHENV | *henv* | input | Environment handle.<br>If *hdbc* is a valid connection handle, *henv* is ignored. |
| SQLHDBC | *hdbc* | input | Database connection handle.<br>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 action for the transaction. The value for this argument must be one of:<br>• SQL_COMMIT<br>• SQL_ROLLBACK |

### Related reference

SQLEndTran() - End transaction of a connection
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.

# Chapter 5. Advanced features

Db2 ODBC provides advanced features for performing setting and retrieving attributes, working with global transactions, querying the catalog, and using LOBs, XML documents, and distinct types.

## Functions for setting and retrieving environment, connection, and statement attributes

Db2 ODBC provides functions that let you set or retrieve a subset of 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 listed below:

- `SQLSetEnvAttr()` - Set environment attributes
- `SQLSetConnectAttr()` - Set connection attributes
- `SQLSetStmtAttr()` - Set statement attributes
- `SQLSetColAttr()` - Set column attributes

Read-only attributes (if any exist) are listed with the detailed function descriptions of the get-attribute functions.

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.

**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.

The following figure shows how you set and retrieve attribute values within a basic connect scenario.

© Copyright IBM Corp. 1997, 2020

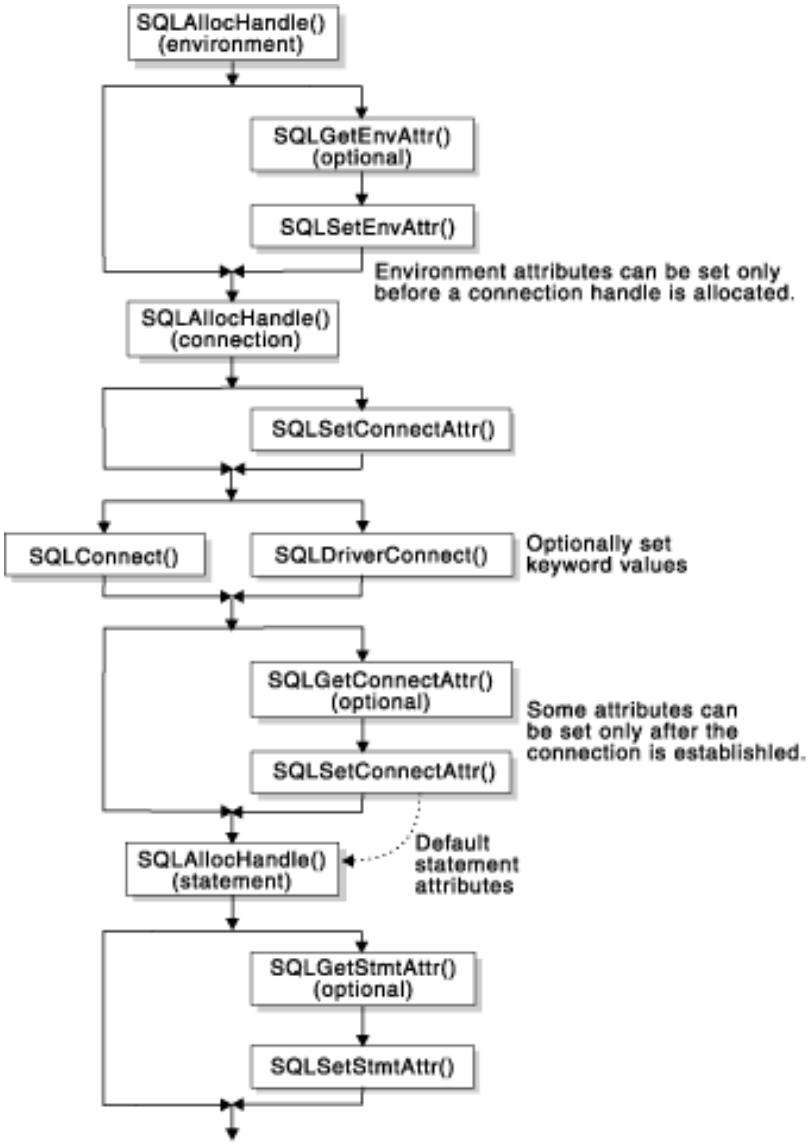



*Figure 37. Setting and retrieving attributes*

**Related concepts**

ODBC programming hints and tips

When you program a Db2 ODBC application, you can avoid common problems, improve performance, reduce network flow, and maximize portability.

Db2 ODBC initialization file

A set of optional keywords can be specified in a Db2 ODBC *initialization file*. An initialization file stores default values for various Db2 ODBC configuration options. Because the initialization file has EBCDIC text, you can use a file editor, such as the TSO editor, to edit it.

**Related reference**

SQLDriverConnect() - Use a connection string to connect to a data source

`SQLDriverConnect()` is an alternative to `SQLConnect()`. Both functions establish a connection to the target database, but `SQLDriverConnect()` supports additional connection parameters.

SQLSetColAttributes() - Set column attributes

`SQLSetColAttributes()` sets the data source result descriptor (column name, type, precision, scale, and nullability) for one column in the result set. If you set the data source result descriptor, Db2 ODBC does not need to obtain the descriptor information from the database management system server.

SQLSetConnectAttr() - Set connection attributes

SQLSetConnectAttr() sets attributes that govern aspects of connections.

SQLSetEnvAttr() - Set environment attributes
SQLSetEnvAttr() sets attributes that affects all connections in an environment.

SQLSetStmtAttr() - Set statement attributes
SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

## Functions for setting and retrieving environment attributes

To specify a new value for an environment attribute, call SQLSetEnvAttr(). To obtain the current value of an environment attribute, call SQLGetEnvAttr().

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.

**Related reference**

SQLGetEnvAttr() - Return current setting of an environment attribute
SQLGetEnvAttr() returns the current setting for an environment attribute. You can also use the SQLSetEnvAttr() function to set these attributes.

SQLSetEnvAttr() - Set environment attributes
SQLSetEnvAttr() sets attributes that affects all connections in an environment.

## Functions for setting and retrieving connection attributes

To specify a new value for a connection attribute, call SQLSetConnectAttr(). To obtain the current value of a connection attribute, call SQLGetConnectAttr().

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

To obtain the current value of a connection attribute, call SQLGetConnectAttr().

**Related reference**

SQLSetConnectAttr() - Set connection attributes
SQLSetConnectAttr() sets attributes that govern aspects of connections.

## Functions for setting and retrieving statement attributes

To specify a new value for a statement attribute, call SQLSetStmtAttr(). To obtain the current value of a statement attribute, call SQLGetStmtAttr().

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

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().

**Related reference**

SQLSetStmtAttr() - Set statement attributes
SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

# ODBC and distributed units of work

You can write Db2 ODBC applications to use distributed units of work.

The transaction scenario that appears in *How to connect to one or more data sources*, 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.

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 for each transaction, as described in *Conceptual view of a Db2 ODBC application*. 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.

**Recommendation:** Set this environment attribute as soon as you successfully allocate an environment handle.

**Related concepts**

How to connect to one or more data sources
Db2 ODBC supports different connection types to remote data sources through DRDA.

Conceptual view of a Db2 ODBC application
A typical Db2 ODBC application includes initialization, transaction processing, and termination tasks.

**Related reference**

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

## Functions for 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.

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.

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.

The following example 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]);
    /*********   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:
```

```
        return (SQL_ERROR);
    } else {
        printf(">Connected to
        return (SQL_SUCCESS);
    }
}
/* ... */
```

*Figure 38. An application that connects to two data sources for a coordinated transaction*

**Related concepts**

Db2 ODBC initialization file

A set of optional keywords can be specified in a Db2 ODBC *initialization file*. An initialization file stores default values for various Db2 ODBC configuration options. Because the initialization file has EBCDIC text, you can use a file editor, such as the TSO editor, to edit it.

## Coordinated connections in a Db2 ODBC application

In distributed units of work, commits and rollbacks among multiple data source connections are coordinated. To establish coordinated connections in a Db2 ODBC application, 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. In a distributed unit of work, you must establish all connections as coordinated. The default commit mode for coordinated connections is manual-commit mode.

The following figure 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. (This figure shows the logical flow and transaction scope of an application that executes the same statements on two SQL_COORDINATED_TRANS connections.)

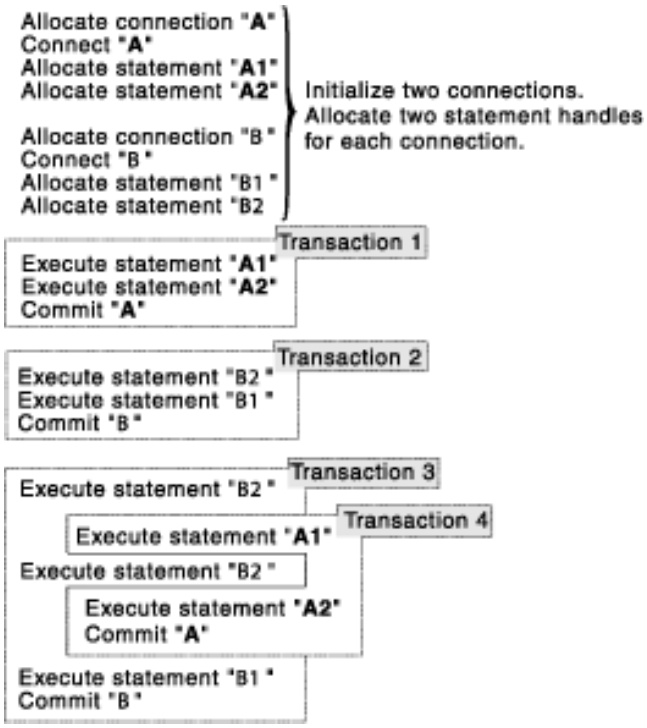



*Figure 39. Multiple connections with concurrent transactions*

In Figure 39 on page 416, 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 39 on page 416, 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 39 on page 416 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.

The following figure shows how the statements that Figure 39 on page 416 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.

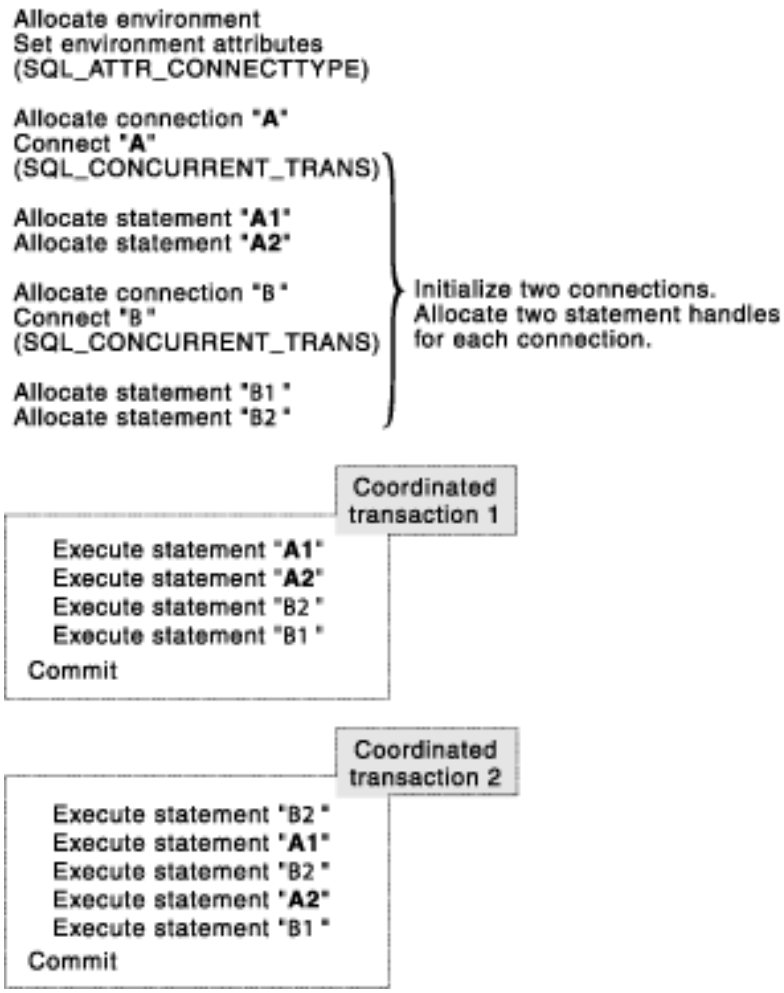



*Figure 40. Multiple connections with coordinated transactions*

**Related concepts**

Commit and rollback in Db2 ODBC

Db2 ODBC supports two commit modes: autocommit and manual-commit. 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.

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.

## Global transactions in ODBC programs

A *global transaction* is a recoverable unit of work, or transaction, that is made up of changes to a collection of resources. 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. 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) coordinate the updates that occur within a global transaction by using a two-phase commit protocol.

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" on page 419 and "4" on page 419 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.

The following example 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, ... );
   .
   .
/*  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 41. An application that performs ODBC global transactions*

**Related reference**

z/OS MVS Programming: Resource Recovery

# Use of ODBC for querying the Db2 catalog

You can use Db2 ODBC catalog query functions and direct catalog queries to the Db2 ODBC shadow catalog to obtain 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.

When you use catalog query functions in your application, queries to the catalog are independent of the way that any single database server 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.

**Related concepts**

Catalog query functions

Catalog functions provide a generic interface for issuing queries and returning consistent result sets across the Db2 servers on different operating systems. In most cases, this consistency allows you to avoid server-specific and release-specific catalog queries in your applications.

The Db2 ODBC shadow catalog

The Db2 ODBC shadow catalog provides increased performance when you need catalog information. To increase the performance of an application that frequently queries the catalog, implement the Db2 ODBC shadow catalog. Redirect catalog functions to the shadow catalog instead of to the native Db2 catalog.

## Catalog query functions

Catalog functions provide a generic interface for issuing queries and returning consistent result sets across the Db2 servers on different operating systems. 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 functions query the catalog and return a result set each:

- “SQLColumnPrivileges() - Get column privileges” on page 138
- “SQLColumns() - Get column information” on page 142
- “SQLForeignKeys() - Get a list of foreign key columns” on page 206
- “SQLPrimaryKeys() - Get primary key columns of a table” on page 323
- “SQLProcedureColumns() - Get procedure input/output parameter information” on page 327
- “SQLProcedures() - Get a list of procedure names” on page 336
- “SQLSpecialColumns() - Get special (row identifier) columns” on page 390
- “SQLStatistics() - Get index and statistics information for a base table” on page 396
- “SQLTablePrivileges() - Get table privileges” on page 401
- “SQLTables() - Get table information” on page 405

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 positions of these columns are 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.

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.

**Related concepts**

Application encoding schemes and Db2 ODBC
Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The Db2 ODBC driver supports input and output character string arguments to ODBC APIs and input and output host variable data in each of these encoding schemes.

**Related reference**

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

**Input arguments on catalog functions**

Input arguments identify or constrain the amount of information that a catalog function returns.

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). *CatalogName*, however, must always be a null pointer (with its length set to 0) because Db2 ODBC does not support three-part naming.

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.

**Related concepts**

Db2 ODBC API entry points

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.

**Related information**

ODBC functions

Db2 ODBC provides various SQL-related functions with unique purposes, diagnostics, and restrictions.

**Catalog functions example**

You can write an application that uses catalog functions to obtain information. For example, you can obtain a list of all tables that have a specified schema name.

The following output shows the 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

```
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 42. Sample output from an application that uses catalog functions*

**Related information**

ODBC functions

Db2 ODBC provides various SQL-related functions with unique purposes, diagnostics, and restrictions.

## The Db2 ODBC shadow catalog

The Db2 ODBC shadow catalog provides increased performance when you need catalog information. To increase the performance of an application that frequently queries the catalog, implement the Db2 ODBC shadow catalog. Redirect catalog functions to the shadow catalog instead of to 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.

Db2 DataPropagator populates and maintains the Db2 ODBC shadow catalog. Db2 for z/OS supports the DATA CAPTURE CHANGE clause of the ALTER TABLE SQL statement. This support allows Db2 to mark log records that are associated with any statements that change the Db2 catalog.

Additionally, the Db2 DataPropagator Capture and Apply process identifies and propagates the Db2 catalog changes to the Db2 ODBC shadow, based on marked log records.

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';
```

The following table lists the base Db2 ODBC shadow catalog tables and the catalog functions that access these tables.

*Table 257. 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() |

*Table 257. Shadow catalog tables and Db2 ODBC APIs (continued)*

| Shadow catalog table | Db2 ODBC catalog function |
|---|---|
| CLISCHEM.SPECIALCOLUMNS | `SQLSpecialColumns()` |
| CLISCHEM.TSTATISTICS | `SQLStatistics()` |
| CLISCHEM.TABLEPRIVILEGES | `SQLTablePrivileges()` |
| CLISCHEM.TABLE | `SQLTables()` |

**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 a set of views of the ODBC shadow catalog base tables.

The following views access 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

## Using arrays to pass parameter values

Db2 ODBC provides an array input method for updating Db2 tables.

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, UPDATE, or MERGE 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 data 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.

   For LOB data in files, you can use `SQLBindFileToParam()`.
2. Call `SQLSetStmtAttr()` 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, update, or merge 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.

**INSERT example:** Consider an application that performs an array insert, as the right side of Figure 43 on page 425 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 SQLSetStmtAttr() with the *crow* argument set to *n*. This value specifies the number of elements in each array.

The following figure shows the two methods of executing a statement with *m* parameters *n* times. Both methods must call SQLBindParameter() once for each parameter.

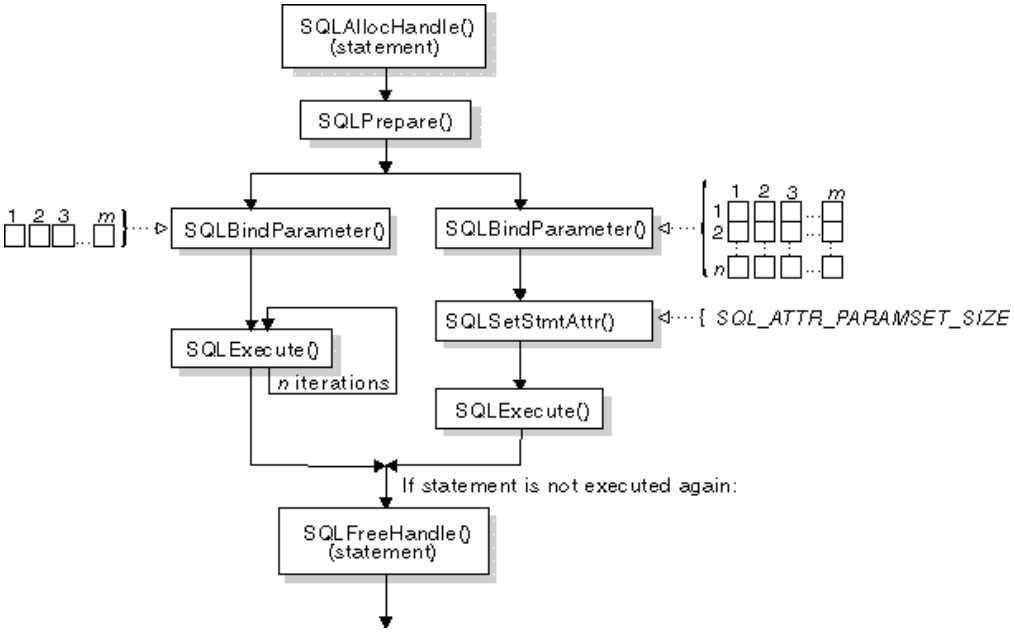



*Figure 43. Array insert*

The left side of the preceding figure 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 on page 425 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 SQLSetStmtAttr() with the statement attribute SQL_ATTR_PARAMSET_SIZE, and then it calls SQLExecute().

The following example shows an array INSERT statement.

```
/* ... */
    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"
    };
```

```
    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
/* ... */
```

**MERGE example:** Consider an application that performs an array merge, as Figure 44 on page 427 illustrates. This application merges an array of EMPNO, FIRSTNME, MIDINIT, LASTNAME, and SALARY values into the DSN8B10.EMP sample table. For each row of values that is to be merged:

- If the EMPNO value for the row that is to be merged matches an EMPNO value in the DSN8B10.EMP table, the SALARY value for the existing table row is updated.
- If the EMPNO value for the row that is to be merged does not match an EMPNO value in the DSN8B10.EMP table, the row is inserted into the DSN8B10.EMP table.

The MERGE statement that accomplishes this is:

```
MERGE INTO DSN8B10.EMP AS t
 USING VALUES
 (CAST (? AS CHAR(6)),
 CAST (? AS VARCHAR(12)),
 CAST (? AS CHAR(1)),
 CAST (? AS VARCHAR(15)),
 CAST (? AS INTEGER))
 AS s (EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY)
 ON t.EMPNO = s.EMPNO
 WHEN MATCHED THEN UPDATE SET SALARY = s.SALARY
 WHEN NOT MATCHED THEN INSERT
 (EMPNO, s.FIRSTNME, s.MIDINIT, s.LASTNAME, s.SALARY)
 NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```

Because this statement contains five parameter markers, the application uses five arrays to store input data. When you make changes to *n* rows, the application places *n* values in each array. When you decide 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 `SQLSetStmtAttr()` with the *crow* argument set to *n*. This value specifies the number of elements in each array.

The following figure shows the two methods of executing a MERGE statement with *m+p* parameters. *m* is the number of parameters that have one or more values for each parameter. *p* is the number of parameters in the UPDATE and INSERT parts of the MERGE statement, which have a single value for each parameter.

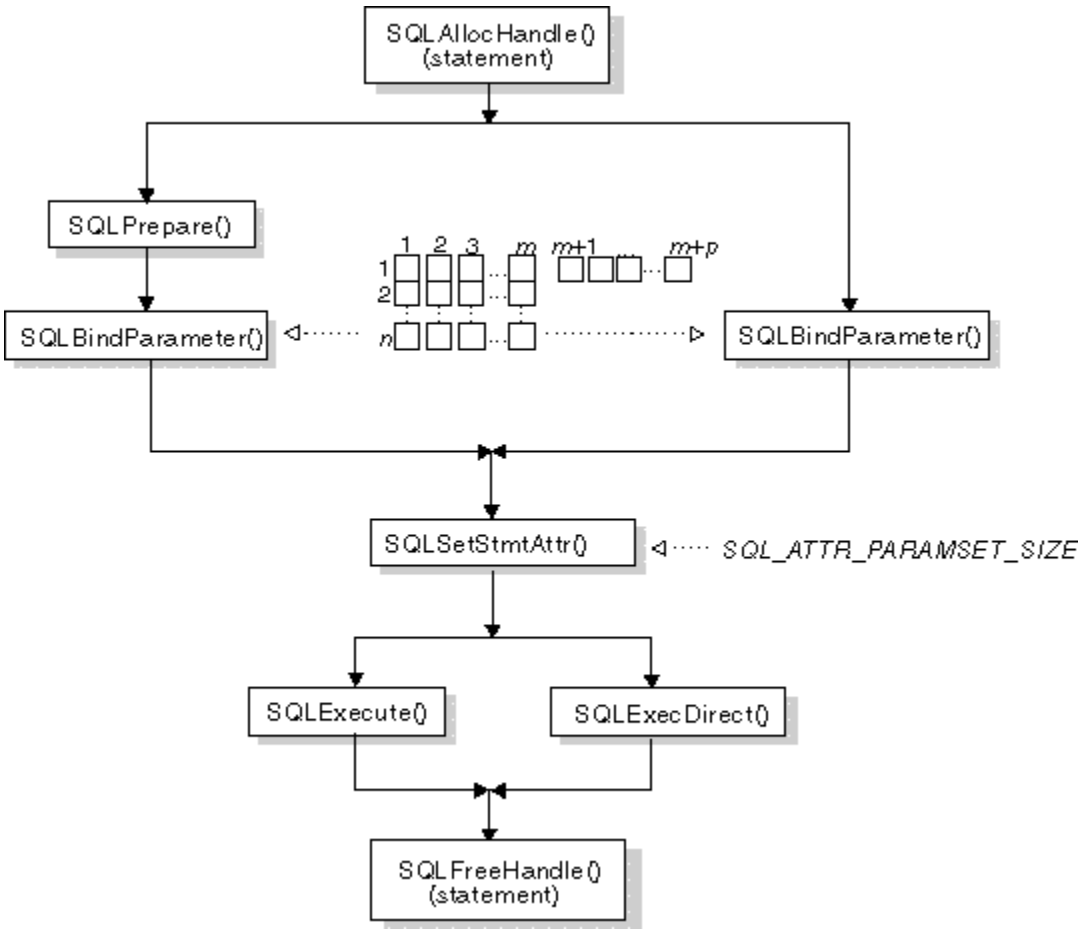



*Figure 44. Array merge*

The left side of the preceding figure illustrates a method of merging you can use when the MERGE statement needs to be executed more than once. The statement is prepared and executed in two separate steps so that the prepared statement can be used if Db2 is set up for prepared statement caching.

The right side of Figure 44 on page 427 illustrates a method of bulk operations that you can use when the MERGE statement needs to be executed only once. The statement is prepared and executed in a single step.

The following figure shows code for an array MERGE.

```
/* declare and initialize local variables */
SQLUINTEGER cRow  = 10;
SQLUINTEGER
piRow = 0;
SQLCHAR     sqlStmt[] =
  "MERGE INTO DSN8B10.EMP AS t"
  " USING VALUES"
  " (CAST (? AS CHAR(6)),"
  " CAST (? AS VARCHAR(12)),"
  " CAST (? AS CHAR(1)),"
  " CAST (? AS VARCHAR(15)),"
  " CAST (? AS INTEGER))"
  " AS s (EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY)"
  " ON t.EMPNO = s.EMPNO"
  " WHEN MATCHED THEN UPDATE SET SALARY = s.SALARY"
  " WHEN NOT MATCHED THEN INSERT"
  " (EMPNO, s.FIRSTNME, s.MIDINIT, s.LASTNAME, s.SALARY)"
  " NOT ATOMIC CONTINUE ON SQLEXCEPTION";
SQLCHAR     empno[10][7];
SQLCHAR     firstname[10][13];
SQLCHAR     middlename[10][2];
SQLCHAR     lastname[10][16];
SQLINTEGER  salary[10];
/* set up data for empno, firstname, middlename, lastname and salary */
/* ... */
/* prepare the statement */
rc = SQLPrepare(hstmt, sqlStmt, SQL_NTS);
/* specify the number of rows to be merged */
rc = SQLParamOptions(hstmt, cRow, &piRow);
/* bind the parameters to input arrays */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                      6, 0, empno, 7, NULL);
rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
                      12, 0, firstname, 13, NULL);
rc = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                      1, 0, middlename, 2, NULL);
rc = SQLBindParameter(hstmt, 4, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
                      15, 0, lastname, 16, NULL);
rc = SQLBindParameter(hstmt, 5, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
                      0, 0, salary, 0, NULL);
/* execute the statement */
rc = SQLExecute(hstmt);
/* display the total number of rows either updated or inserted by MERGE */
printf("MERGEd
```

*Figure 45. An application that performs an array merge*

**Related concepts**

Extended indicators in ODBC applications

ODBC applications can use extended indicators to update all columns in UPDATE, INSERT, and MERGE statements without specifying the current value of columns that do not require changes.

**Related reference**

SQLMoreResults() - Check for more result sets

`SQLMoreResults()` returns more information about a statement handle. The information can be associated with an array of input parameter values for a query, or a stored procedure that returns results sets.

## Retrieval of a result set into an array

An application can issue a query statement and 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.

## Column-wise binding for array data

When you retrieve a result set into an array, you can call `SQLBindCol()` to bind application variables to columns in the result set. Before calling this function, you need to call `SQLSetStmtAttr()` to set the number of rows that you want to retrieve.

The following figure is a logical view of column-wise binding.

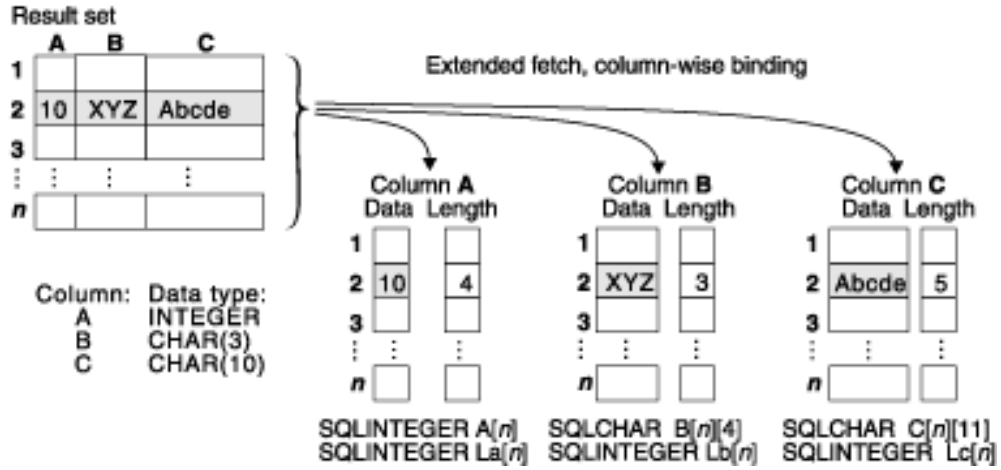



*Figure 46. Column-wise binding*

To perform column-wise array retrieval, include the following procedure in your application:

1. Call `SQLSetStmtAttr()` to set the rowset size.

   If you plan to fetch rows with `SQLExtendedFetch()`, set the SQL_ATTR_ROWSET_SIZE attribute set to the number of rows that you want to retrieve with each fetch.

   If you plan to fetch rows with `SQLFetchScroll()`, set the SQL_ATTR_ROW_ARRAY_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 or SQL_ATTR_ROW_ARRAY_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()` or `SQLFetchScroll()` to retrieve the result data into the array.

   For an `SQLExtendedFetch()` call, 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.

For an SQLExtendedScroll() call, if the number of rows in the result set is greater than the SQL_ATTR_ROW_ARRAY_SIZE attribute value, you must call SQLFetchScroll() 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().

**Related concepts**

Row-wise binding for array data

Row-wise binding associates an entire row of the result set with a structure. You retrieve a rowset 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 only to retrieve data, and not to send it.

## Row-wise binding for array data

Row-wise binding associates an entire row of the result set with a structure. You retrieve a rowset 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 only to retrieve data, and not to send it.

The following figure gives a pictorial view of row-wise binding.

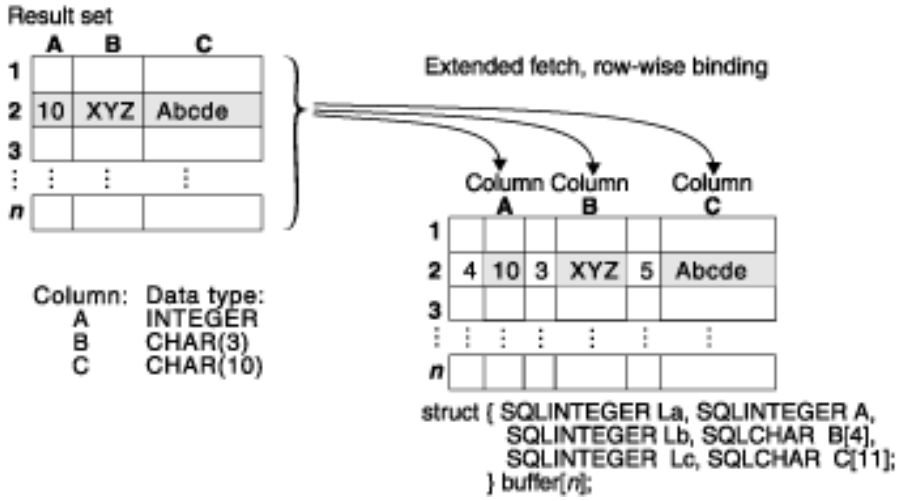



*Figure 47. Row-wise binding*

You must call SQLSetStmtAttr() to set the number of rows that you want to retrieve before you can call SQLBindCol().

To perform row-wise array retrieval, include the following procedure in your application:

1. Call SQLSetStmtAttr() to indicate how many rows to retrieve at a time.

   If you plan to fetch rows with SQLExtendedFetch(), set the SQL_ATTR_ROWSET_SIZE attribute set to the number of rows that you want to retrieve with each fetch.

   If you plan to fetch rows with SQLFetchScroll(), set the SQL_ATTR_ROW_ARRAY_SIZE attribute set to the number of rows that you want to retrieve with each fetch.
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()` or `SQLFetchScroll()` to retrieve the result data into the array.

The following figure shows the required functions to return column-wise and row-wise bound data with `SQLExtendedFetch()`. 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.

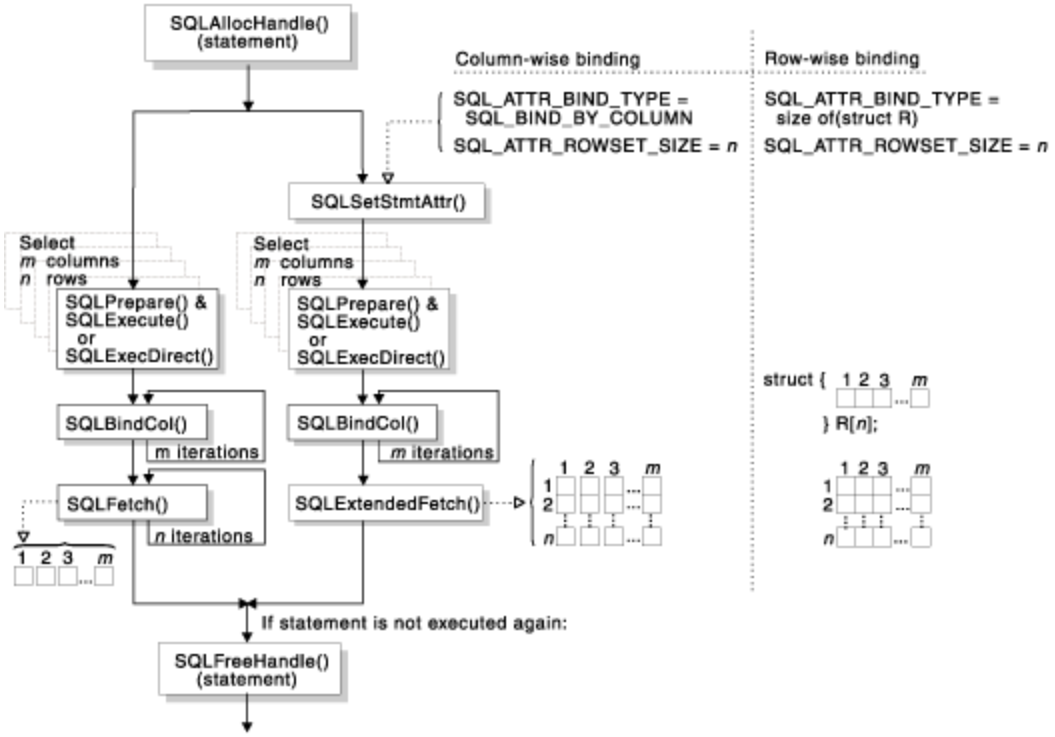



*Figure 48. Array retrieval*

When you perform array retrieval:

- If you specify the value *n* for SQL_ATTR_ROWSET_SIZE (or SQL_ATTR_ROW_ARRAY_SIZE, for `SQLFetchScroll()`), you must retrieve the result set into an array of at least *n* elements. Otherwise, a 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 or SQL_ATTR_ROW_ARRAY_SIZE specifies, you must make multiple calls to `SQLExtendedFetch()` or `SQLFetchScroll()` to retrieve all the rows in the result set. When you make multiple calls to `SQLExtendedFetch()` or `SQLFetchScroll()`, 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*.

**Related concepts**

Retrieval of a result set into an array

An application can issue a query statement and fetch rows from the result set that the query generates.

## The ODBC row status array

The row status array returns the status of each row in the rowset.

You allocate the row status array in your application. Then you specify the address of this array with the SQL_ATTR_ROW_STATUS_PTR statement attribute. The array must have as many elements as are specified by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. `SQLExtendedFetch()`, `SQLFetchScroll()`, or `SQLSetPos()` set the values of the row status array, unless those methods are called after the cursor has been positioned by `SQLExtendedFetch()`. If the value of the SQL_ATTR_ROW_STATUS_PTR statement attribute is a null pointer, `SQLExtendedFetch()`, `SQLFetchScroll()`, and `SQLSetPos()` do not return the row status.

The contents of the row status array buffer are undefined if `SQLExtendedFetch()` or `SQLFetchScroll()` does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

The following values are returned in the row status array.

*Table 258. Values returned in a row status array*

| Row status array value | Description |
|---|---|
| SQL_ROW_SUCCESS | The row was successfully fetched. |
| SQL_ROW_SUCCESS_WITH_INFO | The row was successfully fetched, but a warning was returned about the row. |
| SQL_ROW_ERROR | An error occurred when the row was fetched. |
| SQL_ROW_ADDED | The row was inserted by `SQLBulkOperations()`. If the row is fetched again, or is refreshed by `SQLSetPos()`, its status is SQL_ROW_SUCCESS.<br><br>This value is not set by `SQLExtendedFetch()` or `SQLFetchScroll()`.<br><br>Db2 ODBC does not make inserted rows visible to a scrollable cursor result set, so it does not return this value. |
| SQL_ROW_UPDATED | The row was successfully fetched and has changed since it was last fetched from this result set. If the row is fetched again from this result set, or is refreshed by `SQLSetPos()`, the status changes to the new status for the row.<br><br>Db2 ODBC makes updated rows visible if they continue to satisfy the predicate of the query. Therefore, for `SQLBulkOperations()` or `SQLSetPos()`, Db2 ODBC can indicate SQL_ROW_UPDATED, and the row can be visible upon refetch. However, Db2 ODBC cannot detect a change in value from the last fetch. It returns SQL_SUCCESS for a fetch. |
| SQL_ROW_DELETED | The row was deleted after it was last fetched from this result set.<br><br>Db2 ODBC does not make deleted rows visible to a scrollable cursor result set, so it does not return this value. |

*Table 258. Values returned in a row status array (continued)*

| Row status array value | Description |
|---|---|
| SQL_ROW_NOROW | The rowset overlapped the end of the result set, and no row was returned that corresponds to the corresponding element of the row status array. |

**Related reference**

SQLBulkOperations() - Add, update, delete or fetch a set of rows
`SQLBulkOperations()` adds new rows to the base table or view that is associated with a dynamic cursor for the current query.

SQLFetch() - Fetch the next row
`SQLFetch()` advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLFetchScroll() - Fetch the next row
`SQLFetchScroll()` fetches the specified rowset of data from the result set of a query and returns data for all bound columns. Rowsets can be specified at an absolute position or a relative position.

SQLSetPos - Set the cursor position in a rowset
`SQLSetPos()` sets the cursor position in a rowset. Once the cursor is set, the application can refresh, update, and delete data in the rows.

## Column-wise and row-wise binding example

An application can bind rows and columns of a result set to a structure.

The following example 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 "
     };
/* 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);
    /* 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: 0].Cust_Num);
        printf("************************************\n");
        while (i < pcrow)
        {   printf("\nOrder #: 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("                               j].Total);
        } /* end while */
      } /* end if */
    }
    while ( SQLMoreResults(hstmt) == SQL_SUCCESS);
/* ... */
```

*Figure 49. An application that retrieves data into an array by column and by row*

## ODBC limited block fetch

The Db2 ODBC driver can use limited block fetch to improve performance of FETCH operations on a local Db2 for z/OS server.

With *limited block fetch*, the local Db2 for z/OS server groups the rows that are retrieved by an SQL query into a block of rows in a query buffer. The Db2 ODBC driver retrieves those blocks of rows from the query buffer. Applications that perform single-row fetches or multi-row fetches from large result sets with the `SQLFetch()`, `SQLExtendedFetch()` or `SQLFetchScroll()` function can benefit from limited block fetch. Retrieval of a large number of rows at one time can offer better performance than multiple retrievals of fewer rows.

You can enable limited block fetch without any making any changes to your applications. To enable limited block fetch:

- Set the LIMITEDBLOCKFETCH initialization keyword to 1.

1 is the default value.

- If the default value of 32767 bytes does not provide adequate performance, adjust the QUERYDATASIZE initialization parameter to set the number of bytes that are transferred at one time during FETCH processing. In general, a larger value of QUERYDATASIZE results in fewer trips to the data source, which can result in better performance.

Limited block fetch is effective only for non-scrollable cursors that do not update or delete data.

For applications that use locators when retrieving LOB data from a result set, set LIMITEDBLOCKFETCH to 0. Otherwise, if you attempt to use the SQLGetData() function to retrieve a LOB locator into an application variable after the data has been fetched, the function call fails.

Also, for applications that use the SQL_C_BINARYXML data type, set LIMITEDBLOCKFETCH to 0. Otherwise, if you attempt to use the SQLGetData() function to retrieve XML data and have LIMITEDBLOCKFETCH set to 1, the function call fails.

When you enable limited block fetch, the data that is returned to your application might not reflect the data that has been committed to the source table. For example, suppose that limited block fetch is enabled, and that your application issues `SQLFetch()` to fetch a row from a result set. Db2 ODBC retrieves and stores a block of rows. Suppose that another application concurrently deletes all subsequent rows from the table. The next `SQLFetch()` calls by your application retrieve subsequent rows from the stored block of rows. However, those rows no longer exist in the table. If your application fetches data from tables that are updated by other users, or if your application uses savepoints and issues ROLLBACK TO SAVEPOINT to manage transactions, you should disable limited block fetch.

**Related reference**

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

# Scrollable cursors in Db2 ODBC

Scrollable cursors let you move backward and forward in a query result set.

The ability to scroll back and forth in a result set is essential for applications that need to return to previously fetched rows. Without scrollable cursors, those applications need to implement expensive, cumbersome methods for accessing fetched data again, such as closing and reopening cursors, or caching result sets. Because there is a performance cost for scrollable cursors, you should define scrollable cursors only when your applications require them.

## Scrollable cursor characteristics in Db2 ODBC

In Db2 ODBC, scrollable cursors are defined by whether they are static or dynamic. Scrollable cursors are also defined by their sensitivity, and their concurrency.

### Static or dynamic cursors

Scrollable cursors can be static or dynamic. Static and dynamic cursors differ in their ability to detect updates, deletes, and inserts into the result set. The definitions of static and dynamic cursors are:

**Static**
A read-only cursor. After the cursor is opened, it does not detect any inserts, deletes or updates that are made by its application, or by any other application.

**Dynamic**
A cursor that is sensitive to all inserts, deletes, and updates to the result set that occur after the cursor is opened. A dynamic cursor can insert into, delete from, or update the result set.

### How to set the characteristics of a Db2 ODBC cursor

Before executing a query, an application can specify the cursor type by calling `SQLSetStmtAttr()`, with the statement attribute SQL_ATTR_CURSOR_TYPE. The default cursor type is forward-only.

An application can specify the characteristics of a cursor rather than specifying the cursor type. The application does this by setting the following statement attributes through SQLSetStmtAttr():

- SQL_ATTR_CURSOR_SCROLLABLE
- SQL_ATTR_CURSOR_SENSITIVITY

The ODBC driver selects the cursor type that most efficiently provides the characteristics that the application requests.

Whenever an application sets any of the following statement attributes, the ODBC driver changes the other statement attributes in this set, to keep the behavior consistent:

- SQL_ATTR_CONCURRENCY
- SQL_ATTR_CURSOR_SCROLLABLE
- SQL_TTRCURSOR_SENSITIVITY
- SQL_ATTR_CURSOR_TYPE

The following table lists the default attributes for each cursor type in Db2 ODBC.

*Table 259. Cursor attributes for each cursor type*

| Cursor type | Cursor sensitivity | Cursor concurrency | Cursor scrollability |
|---|---|---|---|
| Forward-only | Unspecified | Read-only concurrency | Not scrollable |
| Static | Insensitive | Read-only concurrency | Scrollable |
| Dynamic | Sensitive | Lock concurrency | Scrollable |

#### How to determine which characteristics are supported

Not all database servers support all types of scrollable cursors. Therefore, before you can use a scrollable cursor, you must determine whether scrollable cursors are supported.

To determine the types of scrollable cursors that are supported by the ODBC driver and the data source, and the capabilities that are supported for each scrollable cursor type, call SQLGetInfo(). Specify the following *InfoType* values:

- SQL_CURSOR_SENSITIVITY
- SQL_SCROLL_OPTIONS
- SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1
- SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2
- SQL_KEYSET_CURSOR_ATTRIBUTES1
- SQL_KEYSET_CURSOR_ATTRIBUTES2
- SQL_DYNAMIC_CURSOR_ATTRIBUTES1
- SQL_DYNAMIC_CURSOR_ATTRIBUTES2

## Relative and absolute scrolling in Db2 ODBC applications

When you use a scrollable cursor, you call SQLFetchScroll() to move the cursor and fetch rows. The set of rows that you fetch is called a *rowset*.

The SQLFetchScroll() function supports relative scrolling (moving to the next row, the previous row, or forward or backward by *n* rows) and absolute scrolling (moving to the first row, the last row, or to row number *n*). The *FetchOrientation* parameter of the SQLFetchScroll() call determines the type of scrolling. Possible values are:

- SQL_FETCH_NEXT
- SQL_FETCH_PRIOR
- SQL_FETCH_FIRST

- SQL_FETCH_LAST
- SQL_FETCH_ABSOLUTE
- SQL_FETCH_RELATIVE

For SQL_FETCH_ABSOLUTE, the *FetchOffset* parameter determines the row to which the cursor moves. For SQL_FETCH_RELATIVE, the *FetchOffset* parameter determines the number of rows by which the cursor moves.

The following figure demonstrates how the cursor moves within a result set for `SQLFetchScroll()` calls with various *FetchOffset* and *FetchOrientation* parameter values. In this example, the rowset size is 3, and the original cursor position is at three rows from the end of the result set.

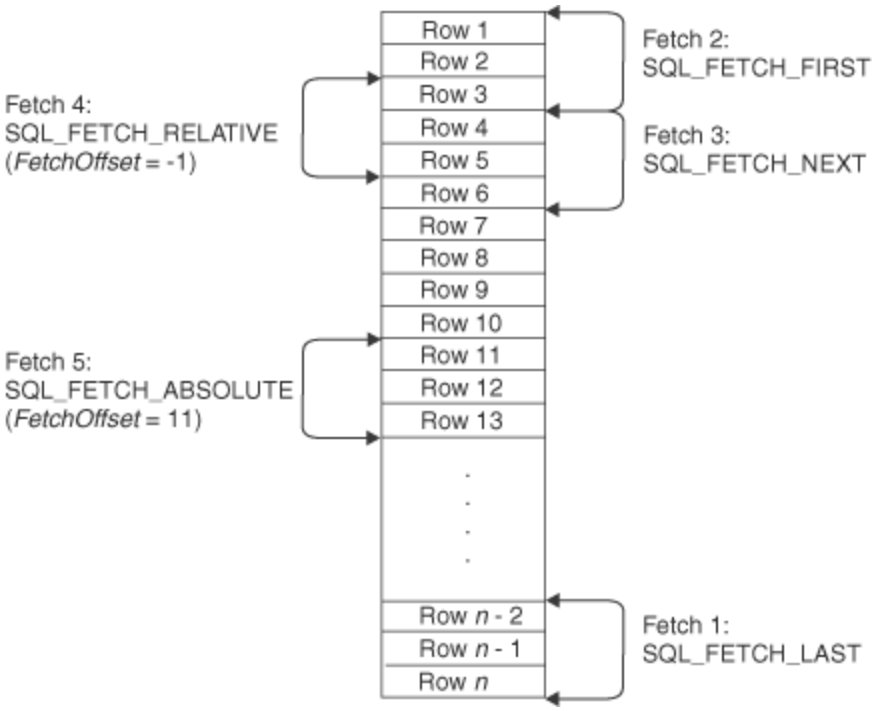



*Figure 50. Example of cursor movement after SQLFetchScroll() calls*

You cannot assume that the entire rowset contains data. Your application must check the rowset size after each fetch, to determine whether the rowset contains a complete set of rows. For example, suppose that you set the rowset size to 10, and you call `SQLFetchScroll()` using A *FetchOrientation* value of SQL_FETCH_ABSOLUTE and a *FetchOffset* value of -3. As the following figure shows, this function call sets the cursor position at three rows from the end of the result set, and attempts to fetch 10 rows.

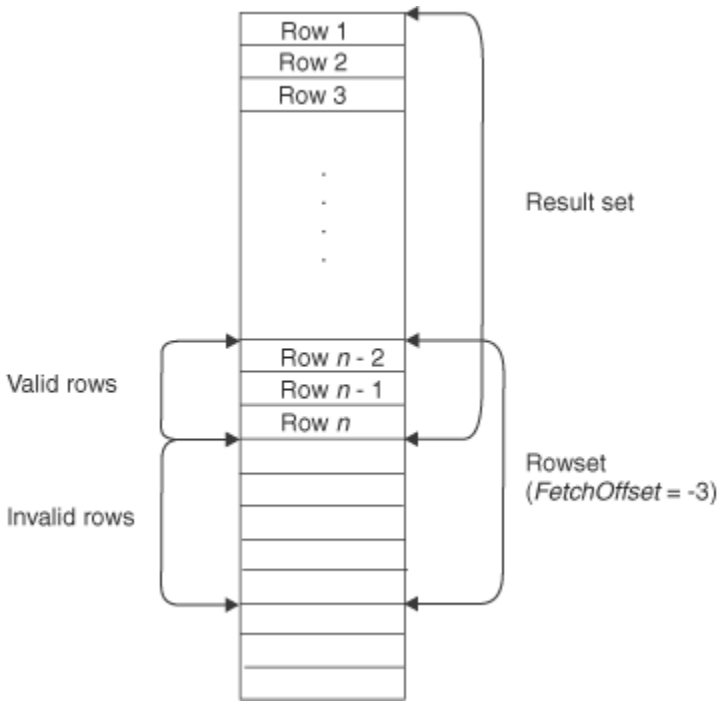



*Figure 51. Example of a SQLFetchScroll() call that returns an incomplete rowset*

After the fetch, only the first three rows of the rowset contain meaningful data, so your application must use the data in only those three rows.

**Related concepts**

The ODBC row status array
The row status array returns the status of each row in the rowset.

**Related reference**

SQLFetchScroll() - Fetch the next row
`SQLFetchScroll()` fetches the specified rowset of data from the result set of a query and returns data for all bound columns. Rowsets can be specified at an absolute position or a relative position.

## Steps for retrieving data with scrollable cursors in a Db2 ODBC application

To use a scrollable cursor in a Db2 ODBC application, you must set the rowset size, specify the scrollable cursor type, set up areas to contain the results of data retrieval, bind the application data, determine the result set size, fetch data, move the cursor multiple times, and close the cursor.

**Procedure**

To retrieve data with scrollable cursors:

1. Call `SQLSetStmtAttr()` to specify the size of the rowset that is returned from the result set.

   Set the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that are returned for each fetch operation. The default rowset size is 1.

   For example, this call declares a rowset size of 35 rows:

   ```
   #define ROWSET_SIZE 35
   /* ... */
   rc = SQLSetStmtAttr(hstmt,
                       SQL_ATTR_ROW_ARRAY_SIZE,
                       (SQLPOINTER) ROWSET_SIZE,
                       0);
   ```

2. Call `SQLSetStmtAttr()` to specify whether to use a static or dynamic scrollable cursor. Set the SQL_ATTR_CURSOR_TYPE statement attribute to SQL_CURSOR_STATIC for a static read-only cursor, or to SQL_CURSOR_DYNAMIC for a dynamic cursor.

The default cursor type is SQL_CURSOR_FORWARD_ONLY.

For example, this call specifies a static cursor:

```
rc = SQLSetStmtAttr(hstmt,
                    SQL_ATTR_CURSOR_TYPE,
                    (SQLPOINTER) SQL_CURSOR_STATIC,
                    0);
```

3. Declare a storage area of type SQLUINTEGER to contain the number or rows that are returned in the rowset from each call to `SQLFetchScroll()`. Call `SQLSetStmtAttr()` to specify the location of that storage area.

   Set the SQL_ATTR_ROWS_FETCHED_PTR statement attribute as a pointer to the storage area.

   For example, this code sets up the rowsFetchedNb variable as the storage area for the number of rows that are returned:

   ```
   /* ... */
   SQLUINTEGER rowsFetchedNb;
   /* ... */
   rc = SQLSetStmtAttr(hstmt,
                       SQL_ATTR_ROWS_FETCHED_PTR,
                       &rowsFetchedNb,
                       0);
   ```

4. Declare a storage area that is an array of SQLUSMALLINT values, to contain the row status array. Call `SQLSetStmtAttr()` to specify the location of the row status array.

   Set the SQL_ATTR_ROW_STATUS_PTR statement attribute as a pointer to the row status array.

   The size of the row status array must be equal to the rowset size that is defined with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.

   For example, this code sets up array row_status as a row status array:

   ```
   /* ... */
   SQLUSMALLINT row_status[ROWSET_SIZE];
   /* ... */
   /* Set a pointer to the array to use for the row status */
   rc = SQLSetStmtAttr(hstmt,
                       SQL_ATTR_ROW_STATUS_PTR,
                       (SQLPOINTER)
                       row_status,
                       0);
   ```

5. Execute a SQL SELECT statement that defines the result set. Bind the results using column-wise or row-wise binding.

   This step is the same as for non-scrollable cursors.

6. Call `SQLRowCount()` to determine the number of rows in the result set.
7. Fetch rowsets of rows from the result set. To do that:
   a) Call `SQLFetchScroll()` to fetch a rowset of data from the result set.

      Set the *FetchOrientation* and *FetchOffset* arguments to indicate the location of the rowset in the result set.

   b) Determine the number of rows that were returned in the result set.

      Db2 ODBC sets this value after each call to `SQLFetchScroll()`, in the location to which statement attribute SQL_ATTR_ROWS_FETCHED_PTR points, and in the row status array.

      For example, this call sets the cursor at the eleventh row of the result set, and fetches a rowset:

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

   c) Display or manipulate the retrieved rows.
   d) Repeat the previous substeps in this step to scroll and fetch more rowsets.

8. After you have retrieved all rowsets, close the cursor by calling SQLCloseCursor(), or free the statement handle by calling SQLFreeHandle() with a *HandleType* value of SQL_HANDLE_STMT. When you free the statement handles, the result set closes.

**Related concepts**

The ODBC row status array
The row status array returns the status of each row in the rowset.

ODBC scrollable cursor example
This ODBC program is an example of how a scrollable cursor can be used to move backward and forward through a result set.

**Related reference**

SQLFetchScroll() - Fetch the next row
SQLFetchScroll() fetches the specified rowset of data from the result set of a query and returns data for all bound columns. Rowsets can be specified at an absolute position or a relative position.

SQLGetInfo() - Get general information
SQLGetInfo() returns general information about the database management systems to which the application is currently connected. For example, SQLGetInfo() indicates which data conversions are supported.

SQLRowCount() - Get row count
SQLRowCount() returns the number of rows in a table that were affected by an UPDATE, INSERT, DELETE, or MERGE statement. You can call SQLRowCount() against a table or against a view that is based on the table. SQLExecute() or SQLExecDirect() must be called before SQLRowCount() is called.

SQLSetStmtAttr() - Set statement attributes
SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

## ODBC scrollable cursor example

This ODBC program is an example of how a scrollable cursor can be used to move backward and forward through a result set.

```
   /*********************************************************************/
   /* 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      i,j,id;
    SQLCHAR         name[51];
    SQLINTEGER      namelen, intlen, colcount;
    struct sqlca    sqlca;
    SQLCHAR   server[18];
    SQLCHAR   uid[30];
    SQLCHAR   pwd[30];
    SQLCHAR   sqlstmt[500];

    SQLINTEGER  H1INT4;
    SQLCHAR     H1CHR10[11];

    SQLINTEGER  LNH1INT4;
```

```
    SQLINTEGER  LNH1CHR10;

    SQLINTEGER output_nts,autocommit,cursor_hold;

    // scrollable cursors
    #define ROWSET_SIZE 10
    SQLUINTEGER     numrowsfetched;
    SQLUSMALLINT    rowStatus[ROWSET_SIZE];
    static char ROWSTATVALUE[][26] =  {  "SQL_ROW_SUCCESS", \
                                         "SQL_ROW_SUCCESS_WITH_INFO", \
                                         "SQL_ROW_ERROR", \
                                         "SQL_ROW_NOROW" };

    // column-wise binding
    SQLINTEGER      SH1INT4[ROWSET_SIZE];
    SQLCHAR         SH1CHR10[ROWSET_SIZE][11];
    SQLINTEGER      SLNH1CHR10[ROWSET_SIZE];

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;}


   /*****************************************************************/
   /* Main Program                                                  */
   /*****************************************************************/
int main()
{

 printf("APDLX INITIALIZATION\n");
//*********************************************************************
 printf("APDLX SQLAllocHandle-Environment\n");
 henv=0;
 rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
 CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc );
 printf("APDLX-henv=%i\n",henv);
//*********************************************************************
 printf("APDLX SQLAllocHandle-Connection\n");
 hdbc=0;
 rc=SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc);
 CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
 printf("APDLX-hdbc=%i\n",hdbc);
//*********************************************************************
 printf("APDLX SQLConnect\n");
 strcpy((char *)uid,"sysadm");
 strcpy((char *)pwd,"sysadm");
 strcpy((char *)server,"stlec1"); //uwo setting
 rc=SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
 CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
 printf("APDLX successfully issued a SQLconnect\n");

//*********************************************************************
 printf("APDLX SQLAllocHandle-Statement\n");
 hstmt=0;
 rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
 printf("APDLX hstmt=%i\n",hstmt);
 printf("APDLX successfully issued a SQLAllocStmt\n");

 /* Set the number of rows in the rowset */
 printf("APDLX SQLSetStmtAttr\n");
 rc = SQLSetStmtAttr( hstmt,
                      SQL_ATTR_ROW_ARRAY_SIZE,
                      (SQLPOINTER) ROWSET_SIZE,
                      0);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );

 /* Set the cursor type */
 printf("APDLX SQLSetStmtAttr\n");
 rc = SQLSetStmtAttr( hstmt,
                      SQL_ATTR_CURSOR_TYPE,
                      (SQLPOINTER) SQL_CURSOR_STATIC,
                      0);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );

 /* Set the pointer to the variable numrowsfetched: */
 printf("APDLX SQLSetStmtAttr\n");
 rc = SQLSetStmtAttr( hstmt,
                      SQL_ATTR_ROWS_FETCHED_PTR,
                      &numrowsfetched,
```

```
                      0);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );

/* Set pointer to the row status array */
printf("APDLX SQLSetStmtAttr\n");
rc = SQLSetStmtAttr( hstmt,
                     SQL_ATTR_ROW_STATUS_PTR,
                     (SQLPOINTER) rowStatus,
                     0);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );

printf("APDLX SQLExecDirect\n");
strcpy((char *)sqlstmt,"SELECT INT4,CHR10 FROM TABLE2A");
printf("APDLX sqlstmt=%s\n",sqlstmt);
rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
printf("APDLX rc=%i\n",rc);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );

printf("APDLX SQLColAttributes\n");
colcount=-1;
rc=SQLColAttributes(hstmt,
                    0,
                    SQL_COLUMN_COUNT,
                    NULL,
                    0,
                    NULL,
                    &colcount);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
if(colcount!=2)
 {
   printf("\nAPDLX colcount=%i\n",colcount);
   goto dberror;
  }

printf("APDLX SQLBindCol\n");
H1INT4=-1;
LNH1INT4=-1;
rc=SQLBindCol(hstmt,
              1,
              SQL_C_LONG,
              (SQLPOINTER) SH1INT4,
              (SQLINTEGER)sizeof(H1INT4),
              (SQLINTEGER *) &LNH1INT4);
if( rc != SQL_SUCCESS ) goto dberror;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );

printf("APDLX SQLBindCol\n");
strcpy(H1CHR10,"garbage");
LNH1CHR10=-1;
rc=SQLBindCol(hstmt,
              2,
              SQL_C_DEFAULT,
              (SQLPOINTER) SH1CHR10,
              11,
              SLNH1CHR10       );
if( rc != SQL_SUCCESS ) goto dberror;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );

printf("\nUse Column-Wise Binding to demonstrate SQLFetchScroll():\n");
printf("\nINT4     CHAR10       \n");
printf("-------- -------------- \n");

printf("APDLX SQLFetchScroll FIRST     \n");
rc = SQLFetchScroll(hstmt, SQL_FETCH_FIRST, 0);
/* Indicate how many rows were in the result set. */
if (rc != SQL_NO_DATA && rc != SQL_SUCCESS)
  CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
printf("(%i rows in rowset). ***\n", numrowsfetched);
for (i = 0; i < numrowsfetched; i++) {
  printf("%8ld %14s\n", SH1INT4[i], SH1CHR10[i]);
}
/* Output the Row Status Array if the complete rowset was not returned. */
if (numrowsfetched != ROWSET_SIZE) {
  printf("  Previous rowset was not full, here is the Row Status Array:\n");
  for (i = 0; i < ROWSET_SIZE; i++)
    printf("    Row Status Array[%i] = %s\n", i, ROWSTATVALUE[rowStatus[i]]);
}

printf("APDLX SQLFetchScroll NEXT     \n");
rc = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);

/* Indicate how many rows were in the result set. */
```

```
if (rc != SQL_NO_DATA && rc != SQL_SUCCESS)
  CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
printf("(%i rows in rowset). ***\n", numrowsfetched);
for (i = 0; i < numrowsfetched; i++) {
  printf("%8ld %14s\n", SH1INT4[i], SH1CHR10[i]);
}
/* Output the Row Status Array if the complete rowset was not returned. */
if (numrowsfetched != ROWSET_SIZE) {
  printf("  Previous rowset was not full, here is the Row Status Array:\n");
  for (i = 0; i < ROWSET_SIZE; i++)
    printf("    Row Status Array[%i] = %s\n", i, ROWSTATVALUE[rowStatus[i]]);
}

printf("APDLX SQLFetchScroll SQL_FETCH_ABSOLUTE 3 \n");
rc = SQLFetchScroll(hstmt, SQL_FETCH_ABSOLUTE, 3);
/* Indicate how many rows were in the result set. */
if (rc != SQL_NO_DATA && rc != SQL_SUCCESS)
  CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
printf("(%i rows in rowset). ***\n", numrowsfetched);
for (i = 0; i < numrowsfetched; i++) {
  printf("%8ld %14s\n", SH1INT4[i], SH1CHR10[i]);
}
/* Output the Row Status Array if the complete rowset was not returned. */
if (numrowsfetched != ROWSET_SIZE) {
  printf("  Previous rowset was not full, here is the Row Status Array:\n");
  for (i = 0; i < ROWSET_SIZE; i++)
    printf("    Row Status Array[%i] = %s\n", i, ROWSTATVALUE[rowStatus[i]]);
}

printf("APDLX SQLFetchScroll SQL_FETCH_RELATIVE -1 \n");
rc = SQLFetchScroll(hstmt, SQL_FETCH_RELATIVE, -1);
/* Indicate how many rows were in the result set. */
if (rc != SQL_NO_DATA && rc != SQL_SUCCESS)
  CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
printf("(%i rows in rowset). ***\n", numrowsfetched);
for (i = 0; i < numrowsfetched; i++) {
  printf("%8ld %14s\n", SH1INT4[i], SH1CHR10[i]);
}
/* Output the Row Status Array if the complete rowset was not returned. */
if (numrowsfetched != ROWSET_SIZE) {
  printf("  Previous rowset was not full, here is the R;ow Status Array:\n");
  for (i = 0; i < ROWSET_SIZE; i++)
    printf("    Row Status Array[%i] = %s\n", i, ROWSTATVALUE[rowStatus[i]]);
}

printf("APDLX SQLFetchScroll SQL_FETCH_FIRST again \n");
rc = SQLFetchScroll(hstmt, SQL_FETCH_FIRST, 0);
printf("rc=%d\n", rc);
/* Indicate how many rows were in the result set. */
if (rc != SQL_NO_DATA && rc != SQL_SUCCESS)
  CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
printf("(%i rows in rowset). ***\n", numrowsfetched);
for (i = 0; i < numrowsfetched; i++) {
  printf("%8ld %14s\n", SH1INT4[i], SH1CHR10[i]);
}
/* Output the Row Status Array if the complete rowset was not returned. */
if (numrowsfetched != ROWSET_SIZE) {
  printf("  Previous rowset was not full, here is the Row Status Array:\n");
  for (i = 0; i < ROWSET_SIZE; i++)
    printf("    Row Status Array[%i] = %s\n", i, ROWSTATVALUE[rowStatus[i]]);
}

printf("APDLX SQLFetchScroll SQL_FETCH_NEXT to EOF \n");
for (j = 0; j < 2; j++) {
  printf("APDLX SQLFetchScroll NEXT     \n");
  rc = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
  printf("rc=%d\n", rc);
  /* Indicate how many rows were in the result set. */
  if (rc != SQL_NO_DATA && rc != SQL_SUCCESS)
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
  printf("(%i rows in rowset). ***\n", numrowsfetched);
  for (i = 0; i < numrowsfetched; i++)
    printf("%8ld %14s\n", SH1INT4[i], SH1CHR10[i]);
  /* Output the Row Status Array if the complete rowset was not returned. */
  if (numrowsfetched != ROWSET_SIZE) {
    printf("  Previous rowset was not full, here is the Row Status Array:\n");
    for (i = 0; i < ROWSET_SIZE; i++)
      printf("    Row Status Array[%i] = %s\n", i, ROWSTATVALUE[rowStatus[i]]);
  }
}  // end for

printf("APDLX SQLFreeHandle-Statement\n");
```

```
 rc=SQLFreeHandle(SQL_HANDLE_STMT,hstmt);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
 hstmt=0;
 printf("APDLX successfully issued a SQLFreeStmt\n");

 printf("APDLX SQLEndTran-Commit\n");
 rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
 CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
 printf("APDLX successfully issued a SQLTransact\n");

 /******** SQLDisconnect ****************************************/
 printf("APDLX SQLDisconnect\n");
 rc=SQLDisconnect(hdbc);
 CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
 printf("APDLX successfully issued a SQLDisconnect\n");

 /******** SQLFreeConnect ***************************************/
 printf("APDLX SQLFreeHandle-Connection\n");
 rc=SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
 CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
 hdbc=0;;
 printf("APDLX successfully issued a SQLFreeConnect\n");

 /******** SQLFreeEnv *******************************************/
 printf("APDLX SQLFreeHandle-Environment\n");
 rc=SQLFreeHandle(SQL_HANDLE_ENV,henv);
 CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc );
 henv=0;
 printf("APDLX successfully issued a SQLFreeEnv\n");

 pgmend:

 printf("APDLX pgmend: Ending sample\n");
 if (rc==0)
  printf("APDLX Execution was SUCCESSFUL\n");
 else
  {
  printf("APDLX*************************\n");
  printf("APDLX Execution FAILED\n");
  printf("APDLX rc = %i\n", rc  );
  printf("APDLX *************************\n");
  }

 return(rc);


 dberror:
 printf("APDLX dberror: entry dberror rtn\n");
 printf("APDLX dberror: rc=%d\n",rc);
 printf("APDLX dberror: environment cleanup attempt\n");
 printf("APDLX dberror: cleanup SQLFreeEnv\n");
 rc=SQLFreeEnv(henv);
 printf("APDLX dberror: cleanup SQLFreeEnv rc =%d\n",rc);
 rc=12;
 printf("APDLX dberror: setting error rc=%d\n",rc);
 goto pgmend;

}  /*END MAIN*/

/*****************************************************************/
/* 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("APDLX entry check_error rtn\n");


    switch (frc) {
    case SQL_SUCCESS:
      break;
    case SQL_INVALID_HANDLE:
      printf("APDLX check_error> SQL_INVALID HANDLE \n");
      break;
    case SQL_ERROR:
      printf("APDLX check_error> SQL_ERROR\n");
      break;
    case SQL_SUCCESS_WITH_INFO:
      printf("APDLX check_error>  SQL_SUCCESS_WITH_INFO\n");
      break;
    case SQL_NO_DATA_FOUND:
      printf("APDLX check_error> SQL_NO_DATA_FOUND\n");
      break;
    default:
      printf("APDLX check_error> Received rc from api rc=%i\n",frc);
      break;
    } /*end switch*/

    print_error(htype,hndl,frc,line,file);

    printf("APDLX SQLGetSQLCA\n");
    rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    if( rc == SQL_SUCCESS )
      prt_sqlca();
    else
      printf("APDLX check_error SQLGetSQLCA failed rc=%i\n",rc);

    printf("APDLX exit check_error rtn\n");
    return (frc);

}  /* end check_error */

/*****************************************************************/
/* 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("APDLX entry print_error rtn\n");

    printf("APDLX 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( "APDLX SQLSTATE: %s\n", sqlstate ) ;
       printf( "APDLX Native Error Code: %ld\n", sqlcode ) ;
       printf( "APDLX buffer: %s \n", buffer ) ;
       i++ ;
    }
    printf( ">-------------------------------------------------\n" ) ;
    printf("APDLX exit print_error rtn\n");
    return( SQL_ERROR ) ;
```

```
}  /* end print_error */

/*********************************************************************/
/* prt_sqlca                                                         */
/*********************************************************************/
SQLRETURN
  prt_sqlca()
  {
    int i;
    printf("APDLX entry prt_sqlca rtn\n");

    printf("\r\rAPDLX*** Printing the SQLCA:\r");
    printf("\nAPDLX SQLCAID .... %s",sqlca.sqlcaid);
    printf("\nAPDLX SQLCABC .... %d",sqlca.sqlcabc);
    printf("\nAPDLX SQLCODE .... %d",sqlca.sqlcode);
    printf("\nAPDLX SQLERRML ... %d",sqlca.sqlerrml);
    printf("\nAPDLX SQLERRMC ... %s",sqlca.sqlerrmc);
    printf("\nAPDLX SQLERRP  ... %s",sqlca.sqlerrp);
    for (i = 0; i < 6; i++)
      printf("\nAPDLX SQLERRD%d ... %d",i+1,sqlca.sqlerrd??(i??));
    for (i = 0; i < 10; i++)
      printf("\nAPDLX SQLWARN%d ... %c",i,sqlca.sqlwarn[i]);
    printf("\nAPDLX SQLWARNA ... %c",sqlca.sqlwarn[10]);
    printf("\nAPDLX SQLSTATE ... %s",sqlca.sqlstate);

    printf("\nAPDLX exit prt_sqlca rtn\n");
    return(0);
  }  /* End of prt_sqlca */
```

*Figure 52. ODBC scrollable cursor example program*

**Related concepts**

The ODBC row status array
The row status array returns the status of each row in the rowset.

**Related tasks**

Steps for retrieving data with scrollable cursors in a Db2 ODBC application
To use a scrollable cursor in a Db2 ODBC application, you must set the rowset size, specify the scrollable cursor type, set up areas to contain the results of data retrieval, bind the application data, determine the result set size, fetch data, move the cursor multiple times, and close the cursor.

**Related reference**

SQLFetchScroll() - Fetch the next row
`SQLFetchScroll()` fetches the specified rowset of data from the result set of a query and returns data for all bound columns. Rowsets can be specified at an absolute position or a relative position.

SQLGetInfo() - Get general information
`SQLGetInfo()` returns general information about the database management systems to which the application is currently connected. For example, `SQLGetInfo()` indicates which data conversions are supported.

SQLRowCount() - Get row count
`SQLRowCount()` returns the number of rows in a table that were affected by an UPDATE, INSERT, DELETE, or MERGE statement. You can call `SQLRowCount()` against a table or against a view that is based on the table. `SQLExecute()` or `SQLExecDirect()` must be called before `SQLRowCount()` is called.

SQLSetStmtAttr() - Set statement attributes

SQLSetStmtAttr() sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call SQLSetConnectAttr().

## Performing bulk inserts with SQLBulkOperations()

You can insert new rows into a table or view at a data source with a call to SQLBulkOperations().

**Before you begin**

Before calling SQLBulkOperations(), an application must ensure that the required bulk operation is supported. To check for support, call SQLGetInfo() with an *InfoType* of SQL_DYNAMIC_CURSOR_ATTRIBUTES1 or SQL_DYNAMIC_CURSOR_ATTRIBUTES2. Check the following attributes to verify that support is available:

- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_BULK_DELETE_BY_BOOKMARK
- SQL_CA1_BULK_FETCH_BY_BOOKMARK

**About this task**

SQLBulkOperations() operates on the current result set through a dynamic cursor, which allows you detect any changes that are made to the result set. SQLBulkOperations() inserts a row using data in the application buffers for each bound column.

**Procedure**

To perform a bulk insert:

1. Execute a query that returns a result set.
2. Set the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that you want to insert.
3. Call SQLBindCol() to bind the data that you want to insert.

   Bind the data to an array with a size that is equal to the value of SQL_ATTR_ROW_ARRAY_SIZE.

   One of the following conditions must be true:

   - The size of the row status array to which the SQL_ATTR_ROW_STATUS_PTR statement attribute points is equal to SQL_ATTR_ROW_ARRAY_SIZE.
   - SQL_ATTR_ROW_STATUS_PTR is a null pointer.

   All bound columns that have a data length of SQL_COLUMN_IGNORE, and all unbound columns must accept NULL values or have a default.
4. Call SQLBulkOperations(*StatementHandle*, SQL_ADD) to perform the insertion.

   SQLBulkOperations() ignores the row operation array to which SQL_ATTR_ROW_OPERATION_PTR points.
5. If the application has set the SQL_ATTR_ROW_STATUS_PTR statement attribute, inspect the row status array to see the result of the operation.

**Example**

The following example is an application that executes a query and uses SQLBulkOperations() to insert 10 rows of data into table CUSTOMER.

```
a#define ROWSET_SIZE 10
/* declare and initialize local variables       */
SQLCHAR sqlstmt[] =
  "SELECT Cust_Num, First_Name, Last_Name FROM CUSTOMER";
SQLINTEGER Cust_Num[ROWSET_SIZE];
SQLCHAR First_Name[ROWSET_SIZE][21];
SQLCHAR Last_Name[ROWSET_SIZE][21];
SQLINTEGER Cust_Num_L[ROWSET_SIZE];
```

```
SQLINTEGER First_Name_L[ROWSET_SIZE];
SQLINTEGER Last_Name_L[ROWSET_SIZE];
SQLUSMALLINT rowStatus[ROWSET_SIZE];
/* Set up dynamic cursor type */
rc = SQLSetStmtAttr(hstmt,
  SQL_ATTR_CURSOR_TYPE,
  (SQLPOINTER) SQL_CURSOR_DYNAMIC,
  0);
/* Set pointer to row status array               */
rc = SQLSetStmtAttr(hstmt,
  SQL_ATTR_ROW_STATUS_PTR,
  (SQLPOINTER) rowStatus,
  0);
/* Execute query */
rc = SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
/* Call SQLBindCol() for each result set column  */
rc = SQLBindCol(hstmt,
  1,
  SQL_C_LONG,
  (SQLPOINTER) Cust_Num,
  (SQLINTEGER) sizeof(Cust_Num)/ROWSET_SIZE,
  Cust_Num_L);
rc = SQLBindCol(hstmt,
  2,
  SQL_C_CHAR,
  (SQLPOINTER) First_Name,
  (SQLINTEGER) sizeof(First_Name)/ROWSET_SIZE,
  First_Name_L);
rc = SQLBindCol(hstmt,
  3,
  SQL_C_CHAR,
  (SQLPOINTER) Last_Name,
  (SQLINTEGER) sizeof(Last_Name)/ROWSET_SIZE,
  Last_Name_L);
…
/* For each column, place the new data values in */
/* the rgbValue array, and set each length value */
/* in the pcbValue array to be the length of the */
/* corresponding value in the rgbValue array.    */
…
/* Set number of rows to insert                  */
rc = SQLSetStmtAttr(hstmt,
  SQL_ATTR_ROW_ARRAY_SIZE,
  (SQLPOINTER) ROWSET_SIZE,
  0);
/* Perform the bulk insert                       */
rc = SQLBulkOperations(hstmt, SQL_ADD);
```

## Updates to Db2 tables with SQLSetPos()

You can update or delete any row in a rowset with a call to SQLSetPos().

SQLSetPos() operates on your current rowset through a dynamic cursor. SQLSetPos() can be used only after a call to SQLFetch() or SQLFetchScroll().

**Related tasks**

Deleting rows in a rowset with SQLSetPos()
You can use SQLSetPos() to delete any row in a rowset. SQLSetPos() operates on your current rowset through a dynamic cursor.

Updating rows in a rowset with SQLSetPos()
You can use SQLSetPos() to update any row in a rowset. SQLSetPos() operates on your current rowset through a dynamic cursor. SQLSetPos() updates a row by using data in the application buffers for each bound column. All bound columns with a data length equal to the value of SQL_COLUMN_IGNORE are not updated, and all unbound columns are not updated.

### Updating rows in a rowset with SQLSetPos()

You can use SQLSetPos() to update any row in a rowset. SQLSetPos() operates on your current rowset through a dynamic cursor. SQLSetPos() updates a row by using data in the application buffers

for each bound column. All bound columns with a data length equal to the value of SQL_COLUMN_IGNORE are not updated, and all unbound columns are not updated.

**Before you begin**

Before you can call SQLSetPos(), you must call SQLFetch() or SQLFetchScroll().

**Procedure**

To update rows with SQLSetPos():

1. Call SQLBindCol() to bind the data that you want to update.
   a) For each bound column, place the new data value in the buffer that is specified by the *rgbValue* argument, and the length of that value in the buffer that is specified by the *pcbValue* argument.
   b) Set the length of the data value of those columns that are not to be updated to SQL_COLUMN_IGNORE.
2. Call SQLSetPos() with *Operation* set to SQL_UPDATE and *RowNumber* set to the number of the row to update.

   If you set *RowNumber* to 0, all rows in the rowset are updated.

   If you set *RowNumber* to 0, but you want to update only certain rows, you can disable the update of the other rows by setting the corresponding elements of the row operation array that is pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute to SQL_ROW_IGNORE.

**Example**

The following example is an application that uses SQLSetPos() to update a row in table CUSTOMER.

```
#define ROWSET_SIZE 10
/* Declare and initialize local variables        */
SQLCHAR sqlstmt[] =
  "SELECT Cust_Num, First_Name, Last_Name FROM CUSTOMER";
SQLINTEGER Cust_Num;
SQLCHAR First_Name[ROWSET_SIZE][21];
SQLCHAR Last_Name[ROWSET_SIZE][21];
SQLINTEGER Cust_Num_L[ROWSET_SIZE];
SQLINTEGER First_Name_L[ROWSET_SIZE];
SQLINTEGER Last_Name_L[ROWSET_SIZE];
SQLUSMALLINT rowStatus[ROWSET_SIZE];
/* Set up dynamic cursor type                    */
rc = SQLSetStmtAttr(hstmt,
  SQL_ATTR_CURSOR_TYPE,
  (SQLPOINTER) SQL_CURSOR_DYNAMIC,
  0);
/* Set pointer to row status array               */
rc = SQLSetStmtAttr(hstmt,
  SQL_ATTR_ROW_STATUS_PTR,
  (SQLPOINTER) rowStatus,
  0);
/* Set number of rows to fetch                   */
rc = SQLSetStmtAttr(hstmt,
  SQL_ATTR_ROW_ARRAY_SIZE,
  (SQLPOINTER) ROWSET_SIZE,
  0);
/* Fetch the first rowset                        */
rc = SQLFetchScroll(hstmt, SQL_FETCH_FIRST, 0);
/* Call SQLBindCol() for each result set column  */
rc = SQLBindCol(hstmt,
  1,
  SQL_C_LONG,
  (SQLPOINTER) Cust_Num,
  (SQLINTEGER) sizeof(Cust_Num)/ROWSET_SIZE,
  Cust_Num_L);
rc = SQLBindCol(hstmt,
  2,
  SQL_C_CHAR,
  (SQLPOINTER) First_Name,
  (SQLINTEGER) sizeof(First_Name)/ROWSET_SIZE,
  First_Name_L);
rc = SQLBindCol(hstmt,
  3,
  SQL_C_CHAR,
```

```
  (SQLPOINTER) Last_Name,
  (SQLINTEGER) sizeof(Last_Name)/ROWSET_SIZE,
  Last_Name_L);
…
/* For each column, place the new data value in  */
/* the rgbValue buffer, and set the length of    */
/* the value in the buffer specified by the      */
/* pcbValue argument                             */
…
/* Update the first row in the rowset            */
rc = SQLSetPos(hstmt, 1, SQL_UPDATE, SQL_LOCK_NO_CHANGE);
```

**Related reference**

SQLBindCol() - Bind a column to an application variable
SQLBindCol() binds a column to an application variable. You can call SQLBindCol() once for each column in a result set from which you want to retrieve data or LOB locators.

SQLFetch() - Fetch the next row
SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns. Columns can be bound to either the application storage or LOB locators.

SQLFetchScroll() - Fetch the next row
SQLFetchScroll() fetches the specified rowset of data from the result set of a query and returns data for all bound columns. Rowsets can be specified at an absolute position or a relative position.

SQLSetPos - Set the cursor position in a rowset
SQLSetPos() sets the cursor position in a rowset. Once the cursor is set, the application can refresh, update, and delete data in the rows.

## Deleting rows in a rowset with SQLSetPos()

You can use SQLSetPos() to delete any row in a rowset. SQLSetPos() operates on your current rowset through a dynamic cursor.

**Before you begin**

Before you can call SQLSetPos(), you must call SQLFetch() or SQLFetchScroll().

**Procedure**

Call SQLSetPos() with *Operation* set to SQL_DELETE and *RowNumber* set to the number of the row to delete.

If you set *RowNumber* to 0, all rows in the rowset are deleted.

If you set *RowNumber* to 0, but you want to delete only certain rows, you can disable the delete of the other rows by setting the corresponding elements of the row operation array that is pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute to SQL_ROW_IGNORE.

**Example**

The following example is an application that uses SQLSetPos() to delete a row in table CUSTOMER.

```
#define ROWSET_SIZE 10
/* declare and initialize local variables        */
SQLCHAR sqlstmt[] =
  "SELECT Cust_Num, First_Name, Last_Name FROM CUSTOMER";
/* Set up dynamic cursor type                    */
rc = SQLSetStmtAttr(hstmt,
  SQL_ATTR_CURSOR_TYPE,
  (SQLPOINTER) SQL_CURSOR_DYNAMIC,
  0);
/* Set pointer to row status array               */
rc = SQLSetStmtAttr(hstmt,
  SQL_ATTR_ROW_STATUS_PTR,
  (SQLPOINTER) rowStatus,
  0);
/* Set number of rows to fetch                   */
rc = SQLSetStmtAttr(hstmt,
  SQL_ATTR_ROW_ARRAY_SIZE,
  (SQLPOINTER) ROWSET_SIZE,
```

```
  0);
/* Fetch first rowset                              */
rc = SQLFetchScroll(hstmt, SQL_FETCH_FIRST, 0);
/* Delete first row in rowset                      */
rc = SQLSetPos(hstmt, 1, SQL_DELETE, SQL_LOCK_NO_CHANGE);
```

## Input and retrieval of long data in pieces

When an application must manipulate long data values, loading the entire values into storage can become impractical. For this reason, Db2 ODBC provides a technique that enables you to handle long data values in pieces.

The technique for handling long data values in pieces is called *specifying parameter values at execute time*. It is the same method that you can use to specify values for fixed-size non-character data types, such as integers.

The following figure 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.

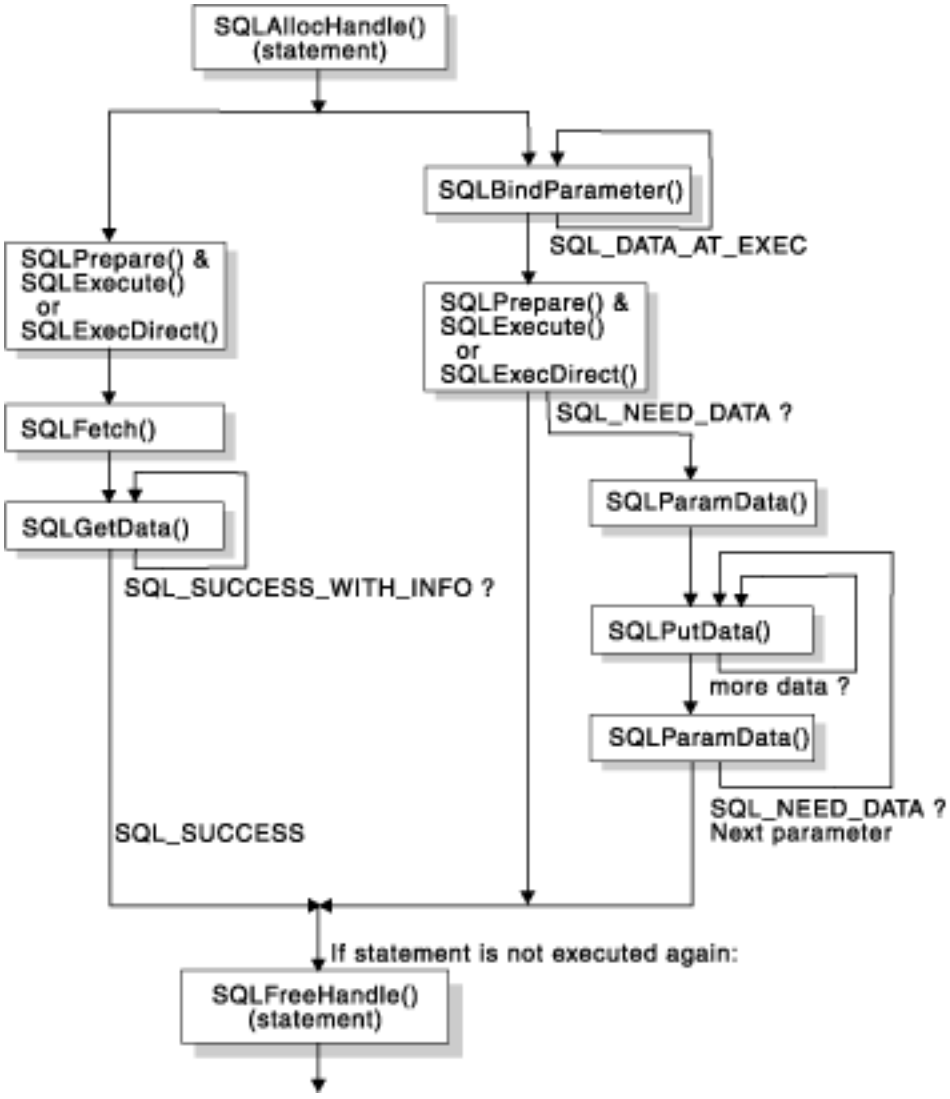



*Figure 53. Input and retrieval of data in pieces*

### Data-at-execute parameters

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” on page 452 and “3” on page 452 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 53 on page 451 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 53 on page 451 without executing the SQL statement.
- SQLGetDiagRec()

You cannot terminate the transaction nor set connection attributes in a data-at-execution flow.

### Data retrieval in pieces

You can call SQLGetData() repeatedly to retrieve smaller pieces of data.

Typically to retrieve data, you allocate application variables to hold the data that 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 53 on page 451 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.

## Providing long data for bulk inserts and positioned updates

To provide long data for bulk inserts or positioned updates, use SQLBulkOperations() or SQLSetPos() calls.

### Procedure

To perform a bulk insert or positioned update with long data:

1. Bind the data using SQLBindCol(). When you call SQLBindCol():
   a) Place an application-defined value, such as the column number, in the **rgbValue* buffer for a data-at-execution column. The value in the **rgbValue* buffer can be used later to identify the column.
   b) Place the SQL_DATA_AT_EXEC value in the **pcbValue* buffer.
2. Call SQLBulkOperations() or SQLSetPos().

   If there are data-at-execution columns, SQLBulkOperations() returns SQL_NEED_DATA and proceeds to step 3. If there are no data-at-execution columns, the process is complete.
3. Call SQLParamData() to retrieve the address of the **rgbValue* buffer for the first data-at-execution column that is to be processed.

   SQLParamData() returns SQL_NEED_DATA.
   a) Retrieve the application-defined value from the **rgbValue* buffer.

      Although data-at-execution parameters are similar to data-at-execution columns, the value that is returned by SQLParamData() is different for data-at-execution parameters and data-at-execution columns. Data-at-execution columns are columns in a rowset for which you send data with SQLPutData() after you insert or update a row with SQLBulkOperations() or SQLSetPos(). You bind data-at-execution columns with SQLBindCol(). The value that is returned by SQLParamData() is the address of the row in the **rgbValue* buffer that is being processed.
4. Call SQLPutData() one or more times to send data for the column.

   More than one call is needed if all the data values cannot be returned in the **rgbValue* buffer that is specified in SQLPutData(). Multiple calls to SQLPutData() for the same column are allowed only under the following circumstances:

   - When you send character C data to a column with a character, binary, or data-source-specific data type
   - When you send binary C data to a column with a character, binary, or data-source-specific data type
5. Call SQLParamData() again to signal that all data has been sent for the column.

   - If there are more data-at-execution columns, SQLParamData() returns SQL_NEED_DATA and the address of the **rgbValue* buffer for the next data-at-execution column that is to be processed. Repeat steps 3 and 3.
   - If there are no more data-at-execution columns, the process is complete.

     If the statement executes successfully, SQLParamData() returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO. If the execution fails, it returns SQL_ERROR. At this point, SQLParamData() can return any SQLSTATE that can be returned by SQLBulkOperations().
   - If the operation is canceled, or an error occurs in SQLParamData() or SQLPutData(), after SQLBulkOperations() or SQLSetPos() returns SQL_NEED_DATA, and before you retrieve data for all data-at-execution columns, you can call only SQLCancel(), SQLGetDiagRec(), SQLGetFunctions(), SQLParamData(), or SQLPutData() for the statement or the connection that is associated with the statement. If you call any other function for the statement or the connection that is associated with the statement, the function returns SQL_ERROR and SQLSTATE HY010 (function sequence error).
   - If you call SQLCancel() while ODBC needs data for data-at-execution columns, ODBC cancels the operation. You can then call SQLBulkOperations() or SQLSetPos() again. Canceling does not affect the cursor state or the current cursor position.

## Examples of using decimal floating point data in an ODBC application

The DECFLOAT(16) and DECFLOAT(34) SQL data types map to the SQLDECIMAL64 and SQLDECIMAL128 C types. You must convert the character data for decimal floating point values to a decimal floating point type before you can assign the data to DECFLOAT columns. You can convert the data that you retrieve from DECFLOAT columns to other data types for manipulation in an application.

The following code demonstrates how to assign decimal floating point data to DECFLOAT columns. The example:

- Assigns string constants that represent decimal floating point values to character variables.
- Converts the character variables to SQLDECIMAL64 and SQLDECIMAL128 format, and assigns the results to SQLDECIMAL64 and SQLDECIMAL128 variables.
- Binds the SQLDECIMAL64 variable to a DECFLOAT(16) parameter marker, and binds the SQLDECIMAL128 variable to a DECFLOAT(34) parameter marker.

```
/* Declare variables for decimal floating point data */
SQLDECIMAL64     H1DFP16;
SQLDECIMAL128 H1DFP34;

SQLINTEGER        LEN_H1DFP16;
SQLINTEGER   LEN_H1DFP34;

SQLCHAR      H1CHAR[100];
decContext_t l_decCtxt64;
decContext_t l_decCtxt128;

/* Initialize a decContext structure to default values */
decContextDefault(&l_decCtxt64, DEC_INIT_DECIMAL64);
decContextDefault(&l_decCtxt128, DEC_INIT_DECIMAL128);

/* Convert a character string to decimal64 format for insert */
strcpy( (char *)H1CHAR, "6400E-2" );
decimal64FromString((decimal64_t *)&H1DFP16,
                    (char *)H1CHAR,
                   &l_decCtxt64);
LEN_H1DFP16 = sizeof(H1DFP16);

/* Convert a character string to decimal128 format for insert */
strcpy( (char *)H1CHAR, "1.28E2" );
decimal128FromString((decimal128_t *)&H1DFP34,
                     (char *)H1CHAR,
                     &l_decCtxt128);
LEN_H1DFP34 = sizeof(H1DFP34);

/* Bind to DECFLOAT(16)*/
SQLParameter( hstmt,
          1,
          SQL_PARAM_INPUT,
          SQL_C_DECIMAL64,
            SQL_DECFLOAT,
              16,
              0,
          (SQLPOINTER)&H1DFP16,
          sizeof(H1DFP16),
          (SQLINTEGER *)&LEN_H1DFP16);

/* Bind to DECFLOAT(34)*/
SQLParameter( hstmt,
          1,
          SQL_PARAM_INPUT,
          SQL_C_DECIMAL128,
            SQL_DECFLOAT,
              34,
              0,
          (SQLPOINTER)&H1DFP34,
          sizeof(H1DFP34),
          (SQLINTEGER *)&LEN_H1DFP34);
```

The following code demonstrates how to retrieve decimal floating point data from DECFLOAT columns. The example:

- Binds a DECFLOAT(16) column to an SQLDECIMAL64 variable, and binds a DECFLOAT(34) column to an SQLDECIMAL128 variable.

- Retrieves the data from the DECFLOAT columns into the SQLDECIMAL64 and SQLDECIMAL128 variables.
- Converts the SQLDECIMAL64 and SQLDECIMAL128 variables to character format, and assigns the results to character variables.

```
/* Declare variables for decimal floating point data */
SQLDECIMAL64     H1DFP16;
SQLDECIMAL128 H1DFP34;

SQLINTEGER         LEN_H1DFP16;
SQLINTEGER    LEN_H1DFP34;

SQLCHAR       H1CHAR[100];
decNumber_t   tempDecNum;


/* Bind DECFLOAT(16) column */
rc = SQLBindCol( hstmt,
                 1,
             SQL_C_DECIMAL64,
             (SQLPOINTER)&H1DFP16,
             sizeof(H1DFP16),
             (SQLINTEGER *)&LEN_H1DFP16);

/* Bind DECFLOAT(34) column */
rc = SQLBindCol( hstmt,
               1,
             SQL_C_DECIMAL128,
             (SQLPOINTER)&H1DFP34,
             sizeof(H1DFP34),
             (SQLINTEGER *)&LEN_H1DFP34);

rc = SQLFetch( hstmt );

/* Convert H1DFP16 to a character string for display */
decimal64ToString( (decimal64_t *)&H1DFP16, (char *)H1CHAR );

/* Convert H1DFP34 to decNumber form in preparation for arithmetic or other operations */
decimal128ToNumber( (decimal128_t *)&H1DFP34, &tempDecNum);
```

## Variable-length timestamps in ODBC applications

For the SQL TIMESTAMP data type, you can specify a timestamp precision of between 0 and 12 digits. You can update timestamps with up to 12 digits of precision in ODBC applications.

The TIMESTAMP data type maps to the C timestamp data types SQL_C_TYPE_TIMESTAMP and SQL_C_TYPE_TIMESTAMP_EXT. You can use SQL_C_TYPE_TIMESTAMP to describe a timestamp with up to 9 fractional digits (nanoseconds). You can use SQL_C_TYPE_TIMESTAMP_EXT to describe a timestamp with up to 12 fractional digits (picoseconds).

The C data type SQL_C_TYPE_TIMESTAMP maps to the C data structure TIMESTAMP_STRUCT. The C data type SQL_C_TYPE_TIMESTAMP_EXT maps to C data structure TIMESTAMP_STRUCT_EXT. Both C data structures contain a `fraction` field that holds the first nine digits of the fractional part of the timestamp, if the timestamp value has a precision of up to nanoseconds. The TIMESTAMP_STRUCT_EXT contains a `fraction2` field that holds the tenth through twelfth digits of the fractional part of the timestamp, if the timestamp value has a precision of up to picoseconds.

When you assign a value to the `fraction` or `fraction2` field, and the length of the value is less than the length of the field, ODBC pads the value on the left to nine digits for `fraction`, and three digits for `fraction2`.

Suppose that a Db2 table is defined like this:

```
CREATE TABLE CLIT1TB1
 (TS0 TIMESTAMP(0),
  TS3 TIMESTAMP(3),
  TS6 TIMESTAMP(9),
  TS9 TIMESTAMP(12),
  TS12 TIMESTAMP(12))
```

The following code demonstrates how to set the `fraction` and `fraction2` fields to insert timestamp values with fractions of seconds into the table.

```
/* Variables for input data                       */
TIMESTAMP_STRUCT_EXT  H1TSTMP0;  /* For timestamp with 0 digits of precision  */
TIMESTAMP_STRUCT_EXT  H1TSTMP3;  /* For timestamp with 3 digits of precision  */
TIMESTAMP_STRUCT_EXT  H1TSTMP9;  /* For timestamp with 9 digits of precision  */
TIMESTAMP_STRUCT_EXT  H1TSTMP12; /* For timestamp with 12 digits of precision */
TIMESTAMP_STRUCT_EXT  H2TSTMP12; /* For timestamp with 12 digits of precision */
/* Variables for input data lengths               */
SQLINTEGER            LEN_H1TSTMP0;
SQLINTEGER            LEN_H1TSTMP3;
SQLINTEGER            LEN_H1TSTMP9;
SQLINTEGER            LEN_H1TSTMP12;
SQLINTEGER            LEN_H2TSTMP12;

/* Set an initial input timestamp value. For     */
/* this example, year, month, day, hour,         */
/* minutes, and seconds are the same for all     */
/* input variables. fraction and fraction2       */
/* vary.                                         */
TIMESTAMP_STRUCT_EXT  input_tstmp =
  {2011, 9, 9, 9, 9, 9, 0, 0 };

/* SQL statement buffer                          */
SQLCHAR     sqlstmt[250];
/* Return code for ODBC calls                    */
SQLRETURN   rc = SQL_SUCCESS;
/* Prepare an INSERT statement for inserting     */
/* data into table CLIT1TB1.                     */
strcpy((char *)sqlstmt,
  "INSERT INTO CLIT1TB1 VALUES (?, ?, ?, ?, ?)");
rc = SQLPrepare( hstmt, sqlstmt, SQL_NTS );
if( rc != SQL_SUCCESS ) goto dberror;

/* Set the first timestamp input host variable   */
/* to this value:                                */
/* 2011-09-09-09.09.09                           */
/* Set fraction and fraction2 to 0 because the   */
/* first timestamp has no fractional portion.    */
H1TSTMP0 = input_tstmp;
H1TSTMP0.fraction=0;
H1TSTMP0.fraction2=0;
/* Set the length of the input host variable     */
LEN_H1TSTMP0 = sizeof(H1TSTMP0);
/* Bind the first timestamp value                */
rc=SQLBindParameter(hstmt,
                    1,
                    SQL_PARAM_INPUT,
                    SQL_C_TYPE_TIMESTAMP_EXT,
                    SQL_TYPE_TIMESTAMP,
                    0,
                    0,
                    &H1TSTMP0,
                    sizeof(H1TSTMP0),
                    &LEN_H1TSTMP0);
if (rc!=SQL_SUCCESS) goto dberror;

/* Set the second timestamp input host variable  */
/* to this value:                                */
/* 2011-09-09-09.09.09.123                       */
/* For the fractional part to be correctly       */
/* interpreted as .123, the fourth through ninth */
/* positions of the fraction field must be set   */
/* to zeros. Set fraction2 to 0 to indicate     */
/* that the timestamp has no picoseconds         */
/* portion.                                      */
H1TSTMP3 = input_tstmp;
H1TSTMP3.fraction=123000000;
H1TSTMP3.fraction2=0;
/* Set the length of the input host variable     */
LEN_H1TSTMP3 = sizeof(H1TSTMP3);
/* Bind the second timestamp value               */
rc=SQLBindParameter(hstmt,
                    2,
                    SQL_PARAM_INPUT,
                    SQL_C_TYPE_TIMESTAMP_EXT,
                    SQL_TYPE_TIMESTAMP,
                    0,
                    3,
```

```
                    &H1TSTMP3,
                    sizeof(H1TSTMP3),
                    &LEN_H1TSTMP3);
if (rc!=SQL_SUCCESS) goto dberror;

/* Set the third timestamp input host variable   */
/* to this value:                                */
/* 2011-09-09-09.09.09.000123456                 */
/* You can omit leading zeros from the          */
/* fractional part of the timestamp. ODBC adds   */
/* the leading zeros for you.                   */
/* Set fraction2 to 0 to indicate that the       */
/* timestamp has no picoseconds portion.         */
H1TSTMP9 = input_tstmp;
H1TSTMP9.fraction=123456;
H1TSTMP9.fraction2=0;
/* Set the length of the input host variable     */
LEN_H1TSTMP9 = sizeof(H1TSTMP9);
/* Bind the third timestamp value                */
rc=SQLBindParameter(hstmt,
                    3,
                    SQL_PARAM_INPUT,
                    SQL_C_TYPE_TIMESTAMP_EXT,
                    SQL_TYPE_TIMESTAMP,
                    0,
                    9,
                    &H1TSTMP9,
                    sizeof(H1TSTMP9),
                    &LEN_H1TSTMP9);
if (rc!=SQL_SUCCESS) goto dberror;

/* Set the fourth timestamp input host variable  */
/* to this value:                                */
/* 2011-09-09-09.09.09.123456789120              */
/* This value has 12 digits of precision, so you */
/* need to set fraction and fraction2.           */
/* Set the first nine fractional digits in       */
/* fraction, and the last three digits in        */
/* fraction2.                                    */
H1TSTMP12 = input_tstmp;
H1TSTMP12.fraction=123456789;
H1TSTMP12.fraction2=120;
/* Set the length of the input host variable     */
LEN_H1TSTMP12 = sizeof(H1TSTMP12);
/* Bind the fourth timestamp value               */
rc=SQLBindParameter(hstmt,
                    4,
                    SQL_PARAM_INPUT,
                    SQL_C_TYPE_TIMESTAMP_EXT,
                    SQL_TYPE_TIMESTAMP,
                    0,
                    12,
                    &H1TSTMP12,
                    sizeof(H1TSTMP12),
                    &LEN_H1TSTMP12);
if (rc!=SQL_SUCCESS) goto dberror;

/* Set the fifth timestamp input host variable   */
/* to this value:                                */
/* 2011-09-09-09.09.09.123456000012              */
/* This value has 12 digits of precision, so you */
/* need to set fraction and fraction2.           */
/* Set the first nine fractional digits in       */
/* fraction, and the last three digits in        */
/* fraction2. You can omit the leading zero from */
/* the value that goes into fraction2.           */
H2TSTMP12 = input_tstmp;
H2TSTMP12.fraction=123456000;
H2TSTMP12.fraction2=12;
/* Set the length of the input host variable     */
LEN_H2TSTMP12 = sizeof(H2TSTMP12);
/* Bind the fifth timestamp value                */
rc=SQLBindParameter(hstmt,
                    5,
                    SQL_PARAM_INPUT,
                    SQL_C_TYPE_TIMESTAMP_EXT,
                    SQL_TYPE_TIMESTAMP,
                    0,
                    12,
                    &H2TSTMP12,
                    sizeof(H2TSTMP12),
                    &LEN_H2TSTMP12);
```

```
if (rc!=SQL_SUCCESS) goto dberror;
/* Execute the INSERT statement                  */
rc=SQLExecute(hstmt);
```

**Related concepts**

Timestamp (Db2 SQL)

**Related reference**

C and SQL data types

Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

# Using LOBs

The term large object (LOB) refers to any type of large object. Db2 supports three LOB data types: binary large object (BLOB), character large object (CLOB), and double-byte character large object (DBCLOB).

These LOB data types are represented symbolically as SQL_BLOB, SQL_CLOB, SQL_DBCLOB respectively. 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.

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.

Another alternative for avoiding the use of application storage is to use file reference variables. With file reference variables, you retrieve LOB values directly from columns into files, or you update LOB column data directly from files.

**Related reference**

C and SQL data types

Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

## Using LOB locators

LOB locators enable you to identify and manipulate LOB values at the database server. They also enable you to retrieve 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:

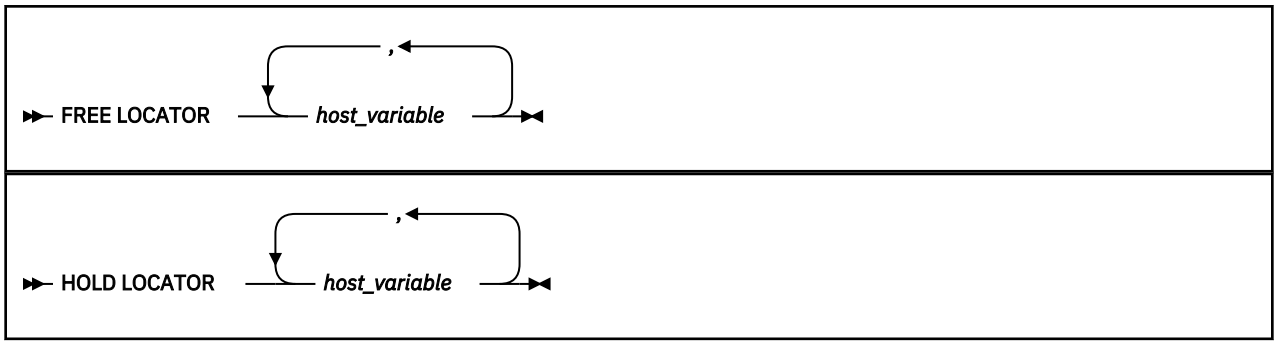
- Fetching a bound LOB column to the appropriate C locator type.
- Calling SQLGetSubString() and specifying that the substring be retrieved as a locator.
- Calling SQLGetData() on an unbound LOB column and specifying the appropriate C locator type. The C locator type must match the LOB column type; otherwise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




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. This calls 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.

For applications that use locators when retrieving LOB data from a result set, set the LIMITEDBLOCKFETCH initialization keyword to 0. Otherwise, if you attempt to use the SQLGetData() function to retrieve a LOB locator into an application variable after the data has been fetched, the function call fails.

**Related reference**

C and SQL data types

Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

## LOB and LOB locator example

An application can use LOB values in the application address space, and LOB locators can help you identify and manipulate these values at the database server.

The following example shows an 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 position */
    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 #:
/* ... */
```

*Figure 54. An application that uses LOB locators*

## LOB file reference variables in ODBC applications

As an alternative to using LOB locators, if an application requires the entire LOB column value, it can request direct file input and output for LOBs. Database queries, updates, and inserts can involve the transfer of single LOB column values into and from files.

The two Db2 ODBC LOB file access functions are:

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

Example:

```
    SQLUINTEGER     fileOption = SQL_FILE_OVERWRITE;
    SQLINTEGER      fileInd = 0;
    SQLSMALLINT     fileNameLength = 14;
    /* ... */
    SQLCHAR         fileName[14] = "LOBFile";
    /* ... */
    rc = SQLBindFileToCol(hstmt, 1, fileName, &fileNameLength,
                          &fileOption, 14, NULL, &fileInd);
```

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

Example:

```
SQLUINTEGER     fileOption = SQL_FILE_OVERWRITE;
SQLINTEGER      fileInd = 0;
SQLSMALLINT     fileNameLength = 14;
/* ... */
SQLCHAR         fileName[14] = "LOBFile";
/* ... */
rc = SQLBindFileToParam(hstmt, 3, SQL_BLOB, fileName,
        &fileNameLength, &fileOption, 14, &fileInd);
```

The file name is the absolute path name of the file. On execute or fetch, data transfer to and from the file occurs, in a similar way to that of bound application variables. A file options argument that is associated with these two functions indicates how the files are to be handled at the time of transfer.

# XML data in ODBC applications

In Db2 tables, the XML built-in data type is used to store XML data in a column as a structured set of nodes in a tree format.

The ODBC symbolic SQL data type SQL_XML corresponds to the Db2 XML data type. The symbolic C data types that you can use for updating XML columns or retrieving data from XML columns are SQL_C_BINARY, SQL_C_CHAR, SQL_C_DBCHAR or SQL_C_WCHAR. The default C data type is SQL_C_BINARY, which is also the recommended data type because it enables the data to be manipulated in its native format. This data type reduces conversion overhead and minimizes data loss that can result from the conversions.

## XML column updates in ODBC applications

When you update or insert data into XML columns of a Db2 table, the input data can be in textual format or Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format (binary XML format)

For XML data, when you use `SQLBindParameter()` or `SQLSetParam()` to bind parameter markers to input data buffers, you can specify the data type of the input data buffer (*fCType*) as one of the following types:

- SQL_C_BINARY
- SQL_C_BINARYXML
- SQL_C_CHAR
- SQL_C_DBCHAR
- SQL_C_WCHAR.

When you bind a data buffer that contains XML data as SQL_C_BINARY, ODBC processes the XML data as internally encoded data. This is the preferred method because it avoids the overhead and potential data loss of character conversion.

**Important:** If the XML data is encoded in an encoding scheme and CCSID other than the application encoding scheme, you need to include internal encoding in the data and bind the data as SQL_C_BINARY to avoid character conversion.

When you bind a data buffer that contains XML data as SQL_C_CHAR, SQL_C_DBCHAR or SQL_C_WCHAR, ODBC processes the XML data as externally encoded data. ODBC determines the encoding of the data as follows:

- If the *fCType* value is SQL_C_WCHAR, ODBC assumes that the data is encoded as UCS-2.
- If the *fCType* value is SQL_C_CHAR or SQL_C_DBCHAR, ODBC assumes that the data is encoded in the application encoding scheme.

SQL_C_BINARYXML is neither internally encoded nor externally encoded. SQL_C_BINARYXML is in binary XML format, as opposed to textual XML format, and it has no encoding.

If you want Db2 to do an implicit XMLPARSE on the data before storing it in an XML column, the parameter marker data type in SQLBindParameter() or SQLSetParam() (*fsqlType*) must be specified as SQL_XML.

If you do an explicit XMLPARSE on the data, the parameter marker data type in SQLBindParameter() or SQLSetParam() (*fsqlType*) can be specified as any character or binary data type.

**Example of inserting XML data into an XML column:** The following example shows how to insert XML data into an XML column by using various C and SQL data types.

```
/* Variables for input XML data                 */
SQLCHAR     HVCHAR[32768];
SQLWCHAR    HVWCHAR[32768];
/* Variables for input XML data lengths         */
SQLINTEGER  LEN_HVCHAR;
SQLINTEGER  LEN_HVWCHAR;
/* SQL statement buffer                         */
SQLCHAR     sqlstmt[250];
/* Return code for ODBC calls                   */
SQLRETURN   rc = SQL_SUCCESS;
/* Prepare an INSERT statement for inserting    */
/* data into an XML column. The input parameter */
/* type is SQL_XML, so DB2 does an implicit     */
/* XMLPARSE.                                    */
strcpy((char *)sqlstmt,
  "INSERT INTO MYTABLE(XMLCOL) VALUES(?)");
/* Bind input XML data with the SQL_C_CHAR type, */
/* to an SQL_XML SQL type.                      */
/* The data is assumed to be externally encoded, */
/* in the application encoding scheme.          */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_XML,
  0, 0, HVCHAR, sizeof(HVCHAR), &LEN_HVCHAR);
/* Execute the INSERT statement */
rc = SQLExecute(hstmt);
/* Bind input XML data with the SQL_C_WCHAR type, */
/* to an SQL_XML SQL type.                       */
/* The data is assumed to be externally encoded,  */
/* in the UCS-2 encoding scheme.                 */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_WCHAR, SQL_XML,
  0, 0, HVWCHAR, sizeof(HVWCHAR0, &LEN_HVWCHAR);
/* Execute the INSERT statement */
rc = SQLExecute(hstmt);
/* Prepare an INSERT statement for inserting    */
/* data into an XML column. The input parameter */
/* type is SQL_CLOB, so the application must    */
/* do an explicit XMLPARSE.                     */
strcpy((char *)sqlstmt,
  "INSERT INTO MYTABLE (XMLCOL) ");
strcat((char *)sqlstmt,
  "VALUES(XMLPARSE(DOCUMENT CAST ? AS CLOB))");
/* Bind input XML data with the SQL_C_CHAR type,  */
/* to an SQL_CLOB SQL type.                      */
/* An explicit XMLPARSE is required for inserting */
/* character data into an XML column.            */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CLOB,
  32768, 0, HVCHAR, sizeof(HVCHAR), &LEN_HVCHAR);
/* Execute the INSERT statement */
rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);
```

*Figure 55. Example of inserting XML data into an XML column*

**Example of inserting binary XML data into an XML column:** The following example shows how to insert binary XML data into an XML column by using the SQL_C_BINARYXML data type.

```
CREATE TABLE MYTABLE ( XML_COL XML );

/* Declare variables for binary XML data */
SQLCHAR HV1BINARYXML[100];
SQLINTEGER LEN_HV1BINARYXML;
SQLCHAR sqlstmt[250];
SQLRETURN rc = SQL_SUCCESS;

/* Assume that HV1BINARYXML contains XML data in binary format
   and LEN_HV1BINARYXML contains the length of data in bytes */

/* Prepare insert statement */
strcpy((char *)sqlstmt, "insert into mytable values(?)");
rc = SQLPrepare(hstmt, sqlstmt, SQL_NTS);

/* Bind XML_COL column as SQL_C_BINARYXML */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARYXML, SQL_XML,
                0, 0, HV1BINARYXML, sizeof(HV1BINARYXML), &LEN_HV1BINARYXML);

/* Execute insert */
rc = SQLExecute(hstmt);
```

*Figure 56. Example of inserting binary XML data into an XML column*

## XML data retrieval in ODBC applications

When you select data from XML columns in a Db2 table, the output data is in textual format or Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format (binary XML format).

For XML data, when you use `SQLBindCol()` to bind columns in a query result set to application variables, you can specify the data type of the application variables (*fCType*) as one of the following types:

- SQL_C_BINARY
- SQL_C_BINARYXML
- SQL_C_CHAR
- SQL_C_DBCHAR
- SQL_C_WCHAR.

The data is returned as internally encoded data.

ODBC determines the encoding of the data as follows:

- If the *fCType* value is SQL_C_BINARY, ODBC returns the data in the UTF-8 encoding scheme.
- If the *fCType* value is SQL_C_BINARYXML, ODBC returns the data in binary XML format.
- If the *fCType* value is SQL_C_CHAR or SQL_C_DBCHAR, ODBC returns the data in the application encoding scheme.
- If the *fCType* value is SQL_C_WCHAR, ODBC returns the data in the UCS-2 encoding scheme.

Db2 performs an implicit XMLSERIALIZE on the data before returning it to your application.

For applications that use the SQL_C_BINARYXML data type, set LIMITEDBLOCKFETCH to 0. Otherwise, if you attempt to use the SQLGetData() function to retrieve XML data and have LIMITEDBLOCKFETCH set to 1, the function call fails.

**Example of retrieving XML data from an XML column:** The following example shows how to retrieve XML data from an XML column into application variables with various C data types.

```
/* Variables for output XML data                   */
SQLCHAR     HVBINARY[32768];
SQLCHAR     HVCHAR[32768];
SQLDBCHAR   HVDBCHAR[32768];
SQLWCHAR    HVWCHAR[32768];
/* Variables for output XML data lengths           */
SQLINTEGER  LEN_HVBINARY;
SQLINTEGER  LEN_HVCHAR;
SQLINTEGER  LEN_HVDBCHAR;
SQLINTEGER  LEN_HVWCHAR;
/* SQL statement buffer                            */
SQLCHAR     sqlstmt[250];
/* Return code for ODBC calls                      */
SQLRETURN   rc = SQL_SUCCESS;
/* Prepare an SELECT statement for retrieving      */
/* data from XML columns.                          */
strcpy((char *)sqlstmt,
  "SELECT XMLCOL1, XMLCOL2, XMLCOL3, XMLCOL4 ");
strcat((char *)sqlstmt,
  "FROM MYTABLE");
/* Bind data for first XML column as SQL_C_BINARY. */
/* This data will be retrieved as internally       */
/* encoded, in the UTF-8 encoding scheme.          */
rc = SQLBindCol(hstmt, 1, SQL_C_BINARY, HVBINARY, sizeof(HVBINARY), &LEN_HVBINARY);
/* Bind data for second XML column as              */
/* SQL_C_CHAR. This data will be retrieved as      */
/* internally encoded, in the application encoding */
/* scheme.                                         */
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, HVCHAR, sizeof(HVCHAR), &LEN_HVCHAR);
/* Bind data for third XML column as SQL_C_DBCHAR. */
/* This data will be retrieved as internally       */
/* encoded, in the application encoding scheme.    */
rc = SQLBindCol(hstmt, 3, SQL_C_DBCHAR, HVDBCHAR, sizeof(HVDBCHAR), &LEN_HVDBCHAR);
/* Bind data for fourth XML column as SQL_C_WCHAR. */
/* This data will be retrieved as internally       */
/* encoded, in the UCS-2 encoding scheme.          */
rc = SQLBindCol(hstmt, 4, SQL_C_WCHAR, HVWCHAR, sizeof(HVWCHAR), &LEN_HVWCHAR);
/* Execute the SELECT statement and fetch a row    */
/* from the result set                             */
rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);
rc = SQLFetch(hstmt);
```

*Figure 57. Example of retrieving XML data from an XML column*

**Example of retrieving binary XML data from an XML column:** The following example shows how to retrieve binary XML data from an XML column into application variables by using type SQL_C_BINARYXML.

```
CREATE TABLE MYTABLE ( XML_COL XML );

/* Declare variables for binary XML data */
SQLCHAR HV1BINARYXML[100];
SQLINTEGER LEN_HV1BINARYXML;
SQLCHAR sqlstmt[250];
SQLRETURN rc = SQL_SUCCESS;

/* Prepare select statement */
strcpy((char *)sqlstmt, "select * from mytable");
rc = SQLPrepare(hstmt, sqlstmt, SQL_NTS);

/* Bind column data as SQL_C_BINARYXML */
rc = SQLBindCol(hstmt, 1, SQL_C_BINARYXML, sizeof(HV1BINARYXML), &LEN_HV1BINARYXML);

/* Execute select */
rc = SQLExecute(hstmt);
/* Fetch result set column as binary XML */
rc = SQLFetch(hstmt);
```

*Figure 58. Example of retrieving binary XML data from an XML column*

## Distinct types in Db2 ODBC applications

You can define your own SQL data type, which is 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.

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 must 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.

The following example shows an application that creates distinct types, user-defined functions, and tables with distinct type columns.

```
/* ... */
    /* 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)",
     "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 stmts);
    /* Execute Direct statements */
    for (i = 0; i < num_stmts; i++) {
        rc = SQLExecDirect(hstmt, stmt[i], SQL_NTS);
    }
/* ... */
```

*Figure 59. An application that creates distinct types*

**Related concepts**

Using arrays to pass parameter values
Db2 ODBC provides an array input method for updating Db2 tables.

Cast parameter markers to distinct types or distinct types to source types
When you use a distinct-type parameter in the predicate of a query statement, you must use a CAST function. You can cast either the parameter marker to a distinct type, or you can cast the distinct type to a source type.

**Related reference**

CREATE TYPE (Db2 SQL)

## Stored procedures for ODBC applications

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.

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. The following figure illustrates this concept.

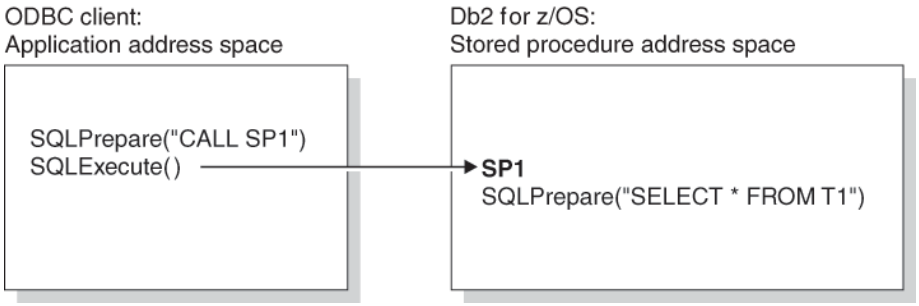



*Figure 60. Running stored procedures*

**Related concepts**

Rules for a Db2 ODBC stored procedure

Db2 ODBC stored procedures are like other Db2 ODBC applications. However, several differences exist.

## Advantages of using stored procedures

With stored procedures, you can avoid network transfer of large amounts of data that is obtained as part of intermediate results in a long sequence of queries. Additionally, you can deploy client database applications as client/server pieces.

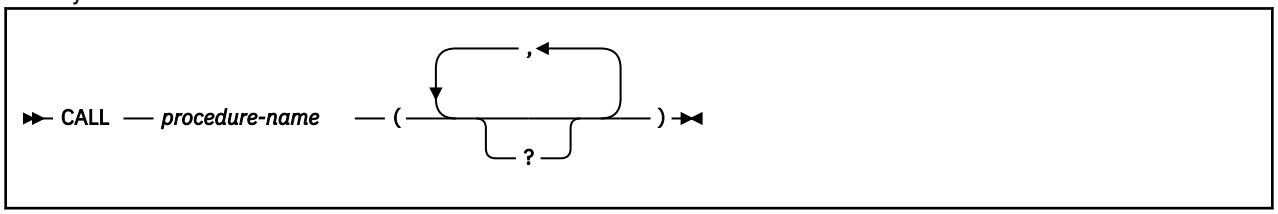
Stored procedures written in embedded static SQL have the following additional advantages:

- 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 must 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 are not required 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.

## Stored procedure calls in a Db2 ODBC application

To invoke stored procedures from a Db2 ODBC application, pass a CALL statement to `SQLExecDirect()`, or to `SQLPrepare()` followed by `SQLExecute()`.

The syntax of the CALL statement is:




***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.

The question mark (?) in the CALL statement syntax diagram denotes parameter markers that correspond to the arguments for a stored procedure. Call `SQLProcedureColumns()` to determine the input and output parameters 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.

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.

**Related concepts**

Vendor escape clauses

Vendor escape clauses increase the portability of your application if your application accesses multiple data sources from different vendors. However, if your application accesses only Db2 data sources, you have no reason to use vendor escape clauses.

**Related reference**

Stored procedure CALL
In ODBC, you can use the extended SQL syntax for calling a stored procedure in a vendor escape clause to make stored procedure calls portable in your SQL statements.

CALL (Db2 SQL)

## Rules for a Db2 ODBC stored procedure

Db2 ODBC stored procedures are like other Db2 ODBC applications. However, several differences exist.

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 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.

## Result sets from stored procedures in ODBC applications

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 various actions.

You can 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.

**Related concepts**

Example Db2 ODBC code

You can view example Db2 ODBC code for a sample verification program DSN8O3VP and for a client application APD29 that calls a stored procedure.

**Restrictions on stored procedures returning result sets**

In general, calling a stored procedure that returns a result set is equivalent to executing a query statement.

Calling a stored procedure that returns a result set is bounded by the following restrictions:

- `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. You receive these result sets in the same way that you receive result sets from a query.

**Procedure**

To write a Db2 ODBC client application that receives result sets from a stored procedure:

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. 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. Use any permitted combination of `SQLBindCol()`, `SQLFetch()`, and `SQLGetData()` to obtain the data set from the current cursor.

   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 470 through "6" on page 470 until you receive all result sets that the stored procedure returned.

**Related reference**

SQLBindCol() - Bind a column to an application variable
`SQLBindCol()` binds a column to an application variable. You can call `SQLBindCol()` once for each column in a result set from which you want to retrieve data or LOB locators.

SQLDescribeCol() - Describe column attributes
`SQLDescribeCol()` returns commonly used descriptor information about a column in a result set that a query generates. Before you call this function, you must call either `SQLPrepare()` or `SQLExecDirect()`.

SQLExecDirect() - Execute a statement directly
`SQLExecDirect()` prepares and executes an SQL statement in one step.

SQLExecute() - Execute a statement
`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.

SQLGetData() - Get data from a column
`SQLGetData()` retrieves data for a single column in the current row of the result set. You can also use `SQLGetData()` to retrieve large data values in pieces. After you call `SQLGetData()` for each column, call `SQLFetch()` or `SQLExtendedFetch()` for each row that you want to retrieve.

SQLMoreResults() - Check for more result sets
`SQLMoreResults()` returns more information about a statement handle. The information can be associated with an array of input parameter values for a query, or a stored procedure that returns results sets.

SQLNumResultCols() - Get number of result columns
`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 you call `SQLNumResultCols()`. After you call `SQLNumResultCols()`, you can call `SQLColAttribute()` or one of the bind column functions.

SQLPrepare() - Prepare a statement

SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the database management system where it is prepared. The application can reference this prepared statement by passing the statement handle to other functions.

# Multithreaded and multiple-context applications in Db2 ODBC

Db2 ODBC supports multi-threading and multiple contexts. You need to follow certain guidelines when using multiple contexts and multi-threading 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.
- For applications that are not HFS-resident, TSO or batch environments that use the Language Environment run time option POSIX(ON).

  **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.
- 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.

Multithreaded applications use threads to perform work in parallel. The following figure depicts an application that performs parallel operations on two different connections and manages a shared application buffer.

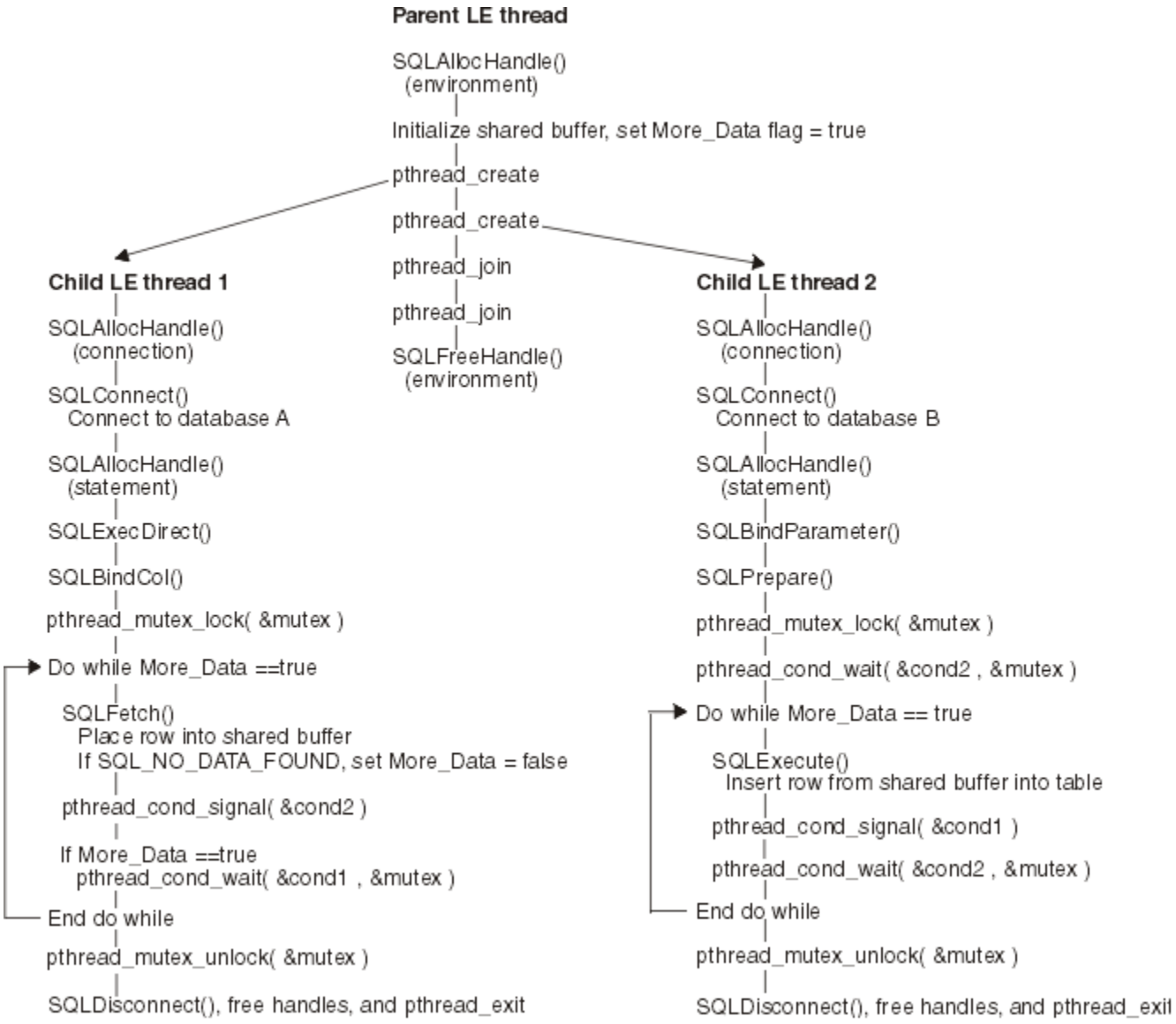



*Figure 61. Multithreaded application*

The application that the preceding figure 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.

**Related concepts**

Overview of preparing and executing a Db2 ODBC application

To prepare and execute a Db2 ODBC application, you need to follow certain steps and understand the Db2 ODBC components.

**Related reference**

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

z/OS UNIX System Services Command Reference

z/OS XL C/C++ runtime library functions (C/C++ Run-Time Library Reference)

**Related information**

Language Environment Programming Guide (z/OS Language Environment Programming Guide)

## When to use multiple Language Environment threads

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.

**Related reference**

z/OS XL 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.

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.

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.

The following table shows the connection characteristics that different combinations of MULTICONTEXT and CONNECTTYPE produce.

*Table 260. Connection characteristics*

| Setting: MULTICONTEXT | Setting: CONNECTTYPE | Result: Language Environment threads can have more than one ODBC connection with an outstanding unit of work | Result: Language Environment threads can commit or roll back an ODBC connection independently | Result: 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 |

**Note:**

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.

**Related concepts**

Db2 ODBC restrictions on the ODBC connection model

Db2 ODBC does not fully support the ODBC connection model if the initialization file does not specify MULTICONTEXT=1.

**Related reference**

SQLGetInfo() - Get general information
SQLGetInfo() returns general information about the database management systems to which the application is currently connected. For example, SQLGetInfo() indicates which data conversions are supported.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

**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.

The following example is 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 62. 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. The following figure shows how an application can fail without a mechanism to serialize use of handles.

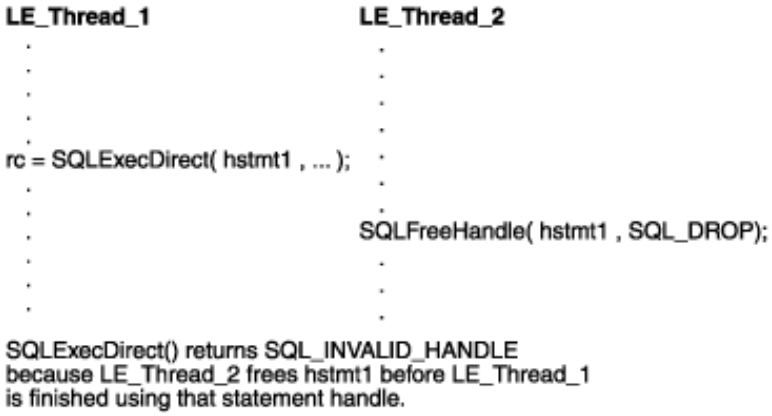



*Figure 63. Example of improper serialization*

The following figure 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.

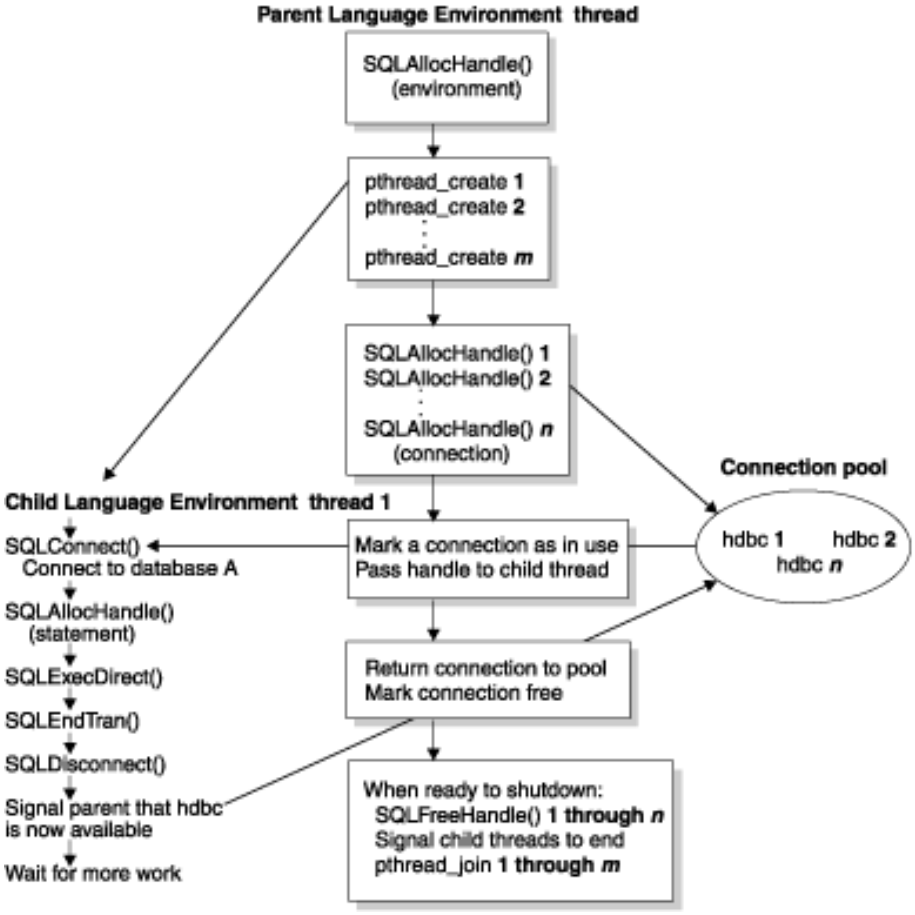



*Figure 64. Model for multithreading with connection pooling (MULTICONTEXT=1)*

To establish a pool of connections (as Figure 64 on page 476 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.

**Related concepts**

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.

## 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()

Db2 ODBC does not support external contexts in applications that run as a stored procedure.

The following example 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 65. An application that manages external contexts*

**Related reference**

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

z/OS MVS Programming: Resource Recovery

MVS Authorized Assembler Services Guide

## 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.

# Application encoding schemes and Db2 ODBC

Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The Db2 ODBC driver supports input and output character string arguments to ODBC APIs 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.

## Types of encoding schemes

Different encoding schemes can represent character data. You can incorporate EBCDIC, ASCII, and Unicode encoding schemes in Db2 ODBC.

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 also accommodate many different languages and geographies.

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 - 6-byte characters)
- UCS-2 (2-byte characters)

## Application programming guidelines for handling different encoding schemes

The Db2 ODBC driver determines whether an application is an EBCDIC, Unicode, or ASCII application by evaluating 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 in the initialization file to EBCDIC, UNICODE, or ASCII, or to a CCSID from which the driver derives the encoding scheme. 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.

#### Db2 ODBC API entry points

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.

  **Example:** To specify a Unicode UCS-2 argument, call a suffix-W API:

```
SQLExecDirectW( (SQLHSTMT) hstmt,
                (SQLWCHAR *) UCS2STR,
               (SQLINTEGER) SQL_NTS );
```

**Related reference**

Suffix-W API function syntax
Db2 for z/OS supports function prototypes for suffix-W APIs, which have slight differences from generic APIs.

**Functions for 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, if the *fCType* argument value is not SQL_C_WCHAR.

  If *fCType* is SQL_C_WCHAR, the encoding scheme is Unicode UCS-2, regardless of the CURRENTAPPENSCH setting.

The following table summarizes how to set the CURRENTAPPENSCH keyword, declare application variables, and declare the *fCType* argument to bind application variables in each encoding scheme.

*Table 261. 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) | Not applicable | UNICODE | ASCII |
| Application variable C type definition | SQLCHAR or SQLDBCHAR | SQLWCHAR | SQLCHAR | SQLCHAR or SQLDBCHAR |
| *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

Db2 for z/OS supports function prototypes for suffix-W APIs, which have slight differences from generic APIs.

The following table compares the function prototypes for suffix-W APIs that Db2 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 262. Comparison of suffix-W APIs to equivalent generic APIs*

| Generic APIs | Suffix-W APIs |
|---|---|
| SQLRETURN **SQLBindFileToCol** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT icol,`<br>`SQLCHAR *FileName,`<br>`SQLSMALLINT *FileNameLength,`<br>`SQLUINTEGER *FileOptions,`<br>`SQLSMALLINT MaxFileNameLength,`<br>`SQLINTEGER *StringLength,`<br>`SQLINTEGER *IndicatorValue);` | SQLRETURN **SQLBindFileToColW** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT icol,`<br>`SQLWCHAR *FileName,`<br>`SQLSMALLINT *FileNameLength,`<br>`SQLUINTEGER *FileOptions,`<br>`SQLSMALLINT MaxFileNameLength,`<br>`SQLINTEGER *StringLength,`<br>`SQLINTEGER *IndicatorValue);` |
| SQLRETURN **SQLBindFileToParam** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT TargetType,`<br>`SQLSMALLINT DataType,`<br>`SQLCHAR *FileName,`<br>`SQLSMALLINT *FileNameLength,`<br>`SQLUINTEGER *FileOptions,`<br>`SQLSMALLINT MaxFileNameLength,`<br>`SQLINTEGER *IndicatorValue);` | SQLRETURN **SQLBindFileToParamW** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT TargetType,`<br>`SQLSMALLINT DataType,`<br>`SQLWCHAR *FileName,`<br>`SQLSMALLINT *FileNameLength,`<br>`SQLUINTEGER *FileOptions,`<br>`SQLSMALLINT MaxFileNameLength,`<br>`SQLINTEGER *IndicatorValue);` |
| SQLRETURN **SQLColAttributes** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT icol,`<br>`SQLUSMALLINT fDescType,`<br>`SQLPOINTER rgbDesc,`<br>`SQLSMALLINT cbDescMax,`<br>`SQLSMALLINT *pcbDesc,`<br>`SQLINTEGER *pfDesc );` | SQLRETURN **SQLColAttributesW** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT icol,`<br>`SQLUSMALLINT fDescType,`<br>`SQLPOINTER rgbDesc,`<br>`SQLSMALLINT cbDescMax,`<br>`SQLSMALLINT *pcbDesc,`<br>`SQLINTEGER *pfDesc );` |
| SQLRETURN **SQLColumns** (<br>`SQLHSTMT hstmt`<br>`SQLCHAR *szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>`SQLCHAR *szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>`SQLCHAR *szTableName,`<br>`SQLSMALLINT cbTableName,`<br>`SQLCHAR *szColumnsName,`<br>`SQLSMALLINT cbColumnName )` | SQLRETURN **SQLColumnsW** (<br>`SQLHSTMT hstmt`<br>**`SQLWCHAR`** `*szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>**`SQLWCHAR`**`*szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>**`SQLWCHAR`** `*szTableName,`<br>`SQLSMALLINT cbTableName,`<br>**`SQLWCHAR`** `*szColumnsName,`<br>`SQLSMALLINT cbColumnName );` |
| SQLRETURN **SQLColumnPrivileges** (<br>`SQLHSTMT hstmt,`<br>`SQLCHAR *szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>`SQLCHAR *szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>`SQLCHAR *szTableName,`<br>`SQLSMALLINT cbTableName,`<br>`SQLCHAR *szColumnsName,`<br>`SQLSMALLINT cbColumnName );` | SQLRETURN **SQLColumnPrivilegesW** (<br>`SQLHSTMT hstmt`<br>**`SQLWCHAR`** `*szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>**`SQLWCHAR`** `*szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>**`SQLWCHAR`** `*szTableName,`<br>`SQLSMALLINT cbTableName,`<br>**`SQLWCHAR`** `*szColumnsName,`<br>`SQLSMALLINT cbColumnName );` |

*Table 262. Comparison of suffix-W APIs to equivalent generic APIs (continued)*

| Generic APIs | Suffix-W APIs |
|---|---|
| SQLRETURN **SQLConnect** (<br>`SQLHDBC hdbc,`<br>`SQLCHAR *szDSN,`<br>`SQLSMALLINT cbDSN,`<br>`SQLCHAR *szUID,`<br>`SQLSMALLINT cbUID,`<br>`SQLCHAR *szAuthStr,`<br>`SQLSMALLINT cbAuthStr );` | SQLRETURN **SQLConnectW** (<br>`SQLHDBC hdbc,`<br>`SQLWCHAR *szDSN,`<br>`SQLSMALLINT cbDSN,`<br>`SQLWCHAR *szUID,`<br>`SQLSMALLINT cbUID,`<br>`SQLWCHAR *szAuthStr,`<br>`SQLSMALLINT cbAuthStr );` |
| SQLRETURN **SQLDataSources** (<br>`SQLHENV henv,`<br>`SQLUSMALLINT fDirection,`<br>`SQLCHAR *szDSN,`<br>`SQLSMALLINT cbDSNMax,`<br>`SQLSMALLINT *pcbDSN,`<br>`SQLCHAR *szDescription,`<br>`SQLSMALLINT cbDescriptionMax,`<br>`SQLSMALLINT *pcbDescription );` | SQLRETURN **SQLDataSourcesW** (<br>`SQLHENV henv,`<br>`SQLUSMALLINT fDirection,`<br>`SQLWCHAR *szDSN,`<br>`SQLSMALLINT cbDSNMax,`<br>`SQLSMALLINT *pcbDSN,`<br>`SQLWCHAR *szDescription,`<br>`SQLSMALLINT cbDescriptionMax,`<br>`SQLSMALLINT *pcbDescription );` |
| SQLRETURN **SQLDescribeCol** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT icol,`<br>`SQLCHAR *szColName,`<br>`SQLSMALLINT cbColNameMax,`<br>`SQLSMALLINT *pcbColName,`<br>`SQLSMALLINT *pfSqlType,`<br>`SQLUINTEGER *pcbColDef,`<br>`SQLSMALLINT *pibScale,`<br>`SQLSMALLINT *pfNullable );` | SQLRETURN **SQLDescribeColW** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT icol,`<br>`SQLWCHAR *szColName,`<br>`SQLSMALLINT cbColNameMax,`<br>`SQLSMALLINT *pcbColName,`<br>`SQLSMALLINT *pfSqlType,`<br>`SQLUINTEGER *pcbColDef,`<br>`SQLSMALLINT *pibScale,`<br>`SQLSMALLINT *pfNullable );` |
| SQLRETURN **SQLDriverConnect** (<br>`SQLHDBC hdbc,`<br>`SQLHWND hwnd,`<br>`SQLCHAR *szConnStrIn,`<br>`SQLSMALLINT cbConnStrIn,`<br>`SQLCHAR *szConnStrOut,`<br>`SQLSMALLINT cbConnStrOutMax,`<br>`SQLSMALLINT pcbConnStrOut,`<br>`SQLUSMALLINT fDriverCompletion );` | SQLRETURN **SQLDriverConnectW** (<br>`SQLHDBC hdbc,`<br>`SQLHWND hwnd,`<br>`SQLWCHAR *szConnStrIn,`<br>`SQLSMALLINT cbConnStrIn,`<br>`SQLWCHAR *szConnStrOut,`<br>`SQLSMALLINT cbConnStrOutMax,`<br>`SQLSMALLINT pcbConnStrOut,`<br>`SQLUSMALLINT fDriverCompletion );` |
| SQLRETURN **SQLError** (<br>`SQLHENV henv,`<br>`SQLHDBC hdbc,`<br>`SQLHSTMT hstmt,`<br>`SQLCHAR *szSqlState,`<br>`SQLINTEGER *pfNativeError,`<br>`SQLCHAR *szErrorMsg,`<br>`SQLSMALLINT cbErrorMsgMax,`<br>`SQLSMALLINT *pcbErrorMsg );` | SQLRETURN **SQLErrorW** (<br>`SQLHENV henv,`<br>`SQLHDBC hdbc,`<br>`SQLHSTMT hstmt,`<br>`SQLWCHAR *szSqlState,`<br>`SQLINTEGER *pfNativeError,`<br>`SQLWCHAR *szErrorMsg,`<br>`SQLSMALLINT cbErrorMsgMax,`<br>`SQLSMALLINT *pcbErrorMsg );` |
| SQLRETURN **SQLExecDirect** (<br>`SQLHSTMT hstmt,`<br>`SQLCHAR *szSqlStr,`<br>`SQLINTEGER cbSqlStr );` | SQLRETURN **SQLExecDirectW** (<br>`SQLHSTMT hstmt,`<br>`SQLWCHAR *szSqlStr,`<br>`SQLINTEGER cbSqlStr );` |

*Table 262. Comparison of suffix-W APIs to equivalent generic APIs (continued)*

| Generic APIs | Suffix-W APIs |
|---|---|
| SQLRETURN **SQLForeignKeys** (<br>`SQLHSTMT hstmt,\|`<br>`SQLCHAR *szPkCatalogName,`<br>`SQLSMALLINT :cbPkCatalogName,`<br>`SQLCHAR *szPkSchemaName,`<br>`SQLSMALLINT :cbPkSchemaName,`<br>`SQLCHAR *szPkTableName,`<br>`SQLSMALLINT :cbPkTableName,`<br>`SQLCHAR *szFkCatalogName,`<br>`SQLSMALLINT :cbFkCatalogName,`<br>`SQLCHAR *szFkSchemaName,`<br>`SQLSMALLINT :cbFkSchemaName,`<br>`SQLCHAR *szFkTableName,`<br>`SQLSMALLINT :cbFkTableName );` | SQLRETURN **SQLForeignKeysW** (<br>`SQLHSTMT hstmt,\|`<br>**`SQLWCHAR`** `*szPkCatalogName,`<br>`SQLSMALLINT :cbPkCatalogName,`<br>**`SQLWCHAR`** `*szPkSchemaName,`<br>`SQLSMALLINT :cbPkSchemaName,`<br>**`SQLWCHAR`** `*szPkTableName,`<br>`SQLSMALLINT :cbPkTableName,`<br>**`SQLWCHAR`** `*szFkCatalogName,`<br>`SQLSMALLINT :cbFkCatalogName,`<br>**`SQLWCHAR`** `*szFkSchemaName,`<br>`SQLSMALLINT :cbFkSchemaName,`<br>**`SQLWCHAR`** `*szFkTableName,`<br>`SQLSMALLINT :cbFkTableName );` |
| SQLRETURN **SQLGetConnectOption** (<br>`SQLHDBC hdbc,`<br>`SQLUSMALLINT fOption,`<br>`SQLUINTEGER pvParam );` | SQLRETURN **SQLGetConnectOptionW** (<br>`SQLHDBC hdbc,`<br>`SQLUSMALLINT fOption,`<br>`SQLUINTEGER pvParam );` |
| SQLRETURN **SQLGetCursorName** (<br>`SQLHSTMT hstmt,`<br>`SQLCHAR *szCursor,`<br>`SQLSMALLINT cbCursorMax,`<br>`SQLSMALLINT *pcbCursor );` | SQLRETURN **SQLGetCursorNameW** (<br>`SQLHSTMT hstmt,`<br>**`SQLWCHAR`** `*szCursor,`<br>`SQLSMALLINT cbCursorMax,`<br>`SQLSMALLINT *pcbCursor );` |
| SQLRETURN **SQLGetInfo** (<br>`SQLHDBC hdbc,`<br>`SQLUSMALLINT fInfoType,`<br>`SQLPOINTER rgbInfoValue,`<br>`SQLSMALLINT cbInfoValueMax,`<br>`SQLSMALLINT *pcbInfoValue );` | SQLRETURN **SQLGetInfoW** (<br>`SQLHDBC hdbc,`<br>`SQLUSMALLINT fInfoType,`<br>`SQLPOINTER rgbInfoValue,`<br>`SQLSMALLINT cbInfoValueMax,`<br>`SQLSMALLINT *pcbInfoValue );` |
| SQLRETURN **SQLGetStmtOption** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT fOption,`<br>`SQLPOINTER pvParam );` | SQLRETURN **SQLGetStmtOptionW**(<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT fOption,`<br>`SQLPOINTER pvParam );` |
| SQLRETURN **SQLGetTypeInfo** (<br>`SQLHSTMT hstmt,`<br>`SQLSMALLINT fSqlType );` | SQLRETURN **SQLGetTypeInfoW** (<br>`SQLHSTMT hstmt,`<br>`SQLSMALLINT fSqlType );` |
| SQLRETURN **SQLNativeSql** (<br>`SQLHDBC hdbc,`<br>`SQLCHAR *szSqlStrIn,`<br>`SQLINTEGER cbSqlStrIn,`<br>`SQLCHAR *szSqlStr,`<br>`SQLINTEGER cbSqlStrMax,`<br>`SQLINTEGER *pcbSqlStr );` | SQLRETURN **SQLNativeSqlW** (<br>`SQLHDBC hdbc,`<br>**`SQLWCHAR`** `*szSqlStrIn,`<br>`SQLINTEGER cbSqlStrIn,`<br>**`SQLWCHAR`** `*szSqlStr,`<br>`SQLINTEGER cbSqlStrMax,`<br>`SQLINTEGER *pcbSqlStr );` |
| SQLRETURN **SQLPrepare** (<br>`SQLHSTMT hstmt,`<br>`SQLCHAR *szSqlStr,`<br>`SQLINTEGER cbSqlStr );` | SQLRETURN **SQLPrepareW** (<br>`SQLHSTMT hstmt,`<br>**`SQLWCHAR`** `*szSqlStr,`<br>`SQLINTEGER cbSqlStr );` |

*Table 262. Comparison of suffix-W APIs to equivalent generic APIs (continued)*

| Generic APIs | Suffix-W APIs |
|---|---|
| SQLRETURN **SQLPrimaryKeys** (<br>`SQLHSTMT hstmt,`<br>`SQLCHAR *szCatalogName,`<br>`SQLSMALLINT :cbCatalogName,`<br>`SQLCHAR *szSchemaName,`<br>`SQLSMALLINT :cbSchemaName,`<br>`SQLCHAR *szTableName,`<br>`SQLSMALLINT :cbTableName );` | SQLRETURN **SQLPrimaryKeysW** (<br>`SQLHSTMT hstmt,`<br>`SQLWCHAR *szCatalogName,`<br>`SQLSMALLINT :cbCatalogName,`<br>`SQLWCHAR *szSchemaName,`<br>`SQLSMALLINT :cbSchemaName,`<br>`SQLWCHAR *szTableName,`<br>`SQLSMALLINT :cbTableName );` |
| SQLRETURN **SQLProcedureColumns** (<br>`SQLHSTMT hstmt,`<br>`SQLCHAR *szProcCatalog,`<br>`SQLSMALLINT cbProcCatalog,`<br>`SQLCHAR *szProcSchema,`<br>`SQLSMALLINT cbProcSchema,`<br>`SQLCHAR *szProcName,`<br>`SQLSMALLINT cbProcName,`<br>`SQLCHAR *szColumnName,`<br>`SQLSMALLINT cbColumnName );` | SQLRETURN **SQLProcedureColumnsW** (<br>`SQLHSTMT hstmt,`<br>`SQLWCHAR *szProcCatalog,`<br>`SQLSMALLINT cbProcCatalog,`<br>`SQLWCHAR *szProcSchema,`<br>`SQLSMALLINT cbProcSchema,`<br>`SQLWCHAR *szProcName,`<br>`SQLSMALLINT cbProcName,`<br>`SQLWCHAR *szColumnName,`<br>`SQLSMALLINT cbColumnName );` |
| SQLRETURN **SQLProcedures** (<br>`SQLHSTMT hstmt,`<br>`SQLCHAR *szProcCatalog,`<br>`SQLSMALLINT cbProcCatalog,`<br>`SQLCHAR *szProcSchema,`<br>`SQLSMALLINT cbProcSchema,`<br>`SQLCHAR *szProcName,`<br>`SQLSMALLINT cbProcName );` | SQLRETURN **SQLProceduresW** (<br>`SQLHSTMT hstmt,`<br>`SQLWCHAR *szProcCatalog,`<br>`SQLSMALLINT cbProcCatalog,`<br>`SQLWCHAR *szProcSchema,`<br>`SQLSMALLINT cbProcSchema,`<br>`SQLWCHAR *szProcName,`<br>`SQLSMALLINT cbProcName );` |
| SQLRETURN **SQLSetConnectOption** (<br>`SQLHDBC hdbc,`<br>`SQLUSMALLINT fOption,`<br>`SQLPOINTER pvParam );` | SQLRETURN **SQLSetConnectOptionW** (<br>`SQLHDBC hdbc,`<br>`SQLUSMALLINT fOption,`<br>`SQLPOINTER pvParam );` |
| SQLRETURN **SQLSetCursorName** (<br>`SQLHSTMT hstmt,`<br>`SQLCHAR *szCursor,`<br>`SQLSMALLINT cbCursor );` | SQLRETURN **SQLSetCursorNameW** (<br>`SQLHSTMT hstmt,`<br>`SQLWCHAR *szCursor,`<br>`SQLSMALLINT cbCursor );` |
| SQLRETURN **SQLSetStmtOption** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT fOption`<br>`SQLUINTEGER pvParam );` | SQLRETURN **SQLSetStmtOptionW** (<br>`SQLHSTMT hstmt,`<br>`SQLUSMALLINT fOption`<br>`SQLUINTEGER pvParam );` |
| SQLRETURN **SQLSpecialColumns** (<br>`SQLHSTMT hstmt`<br>`SQLUSMALLINT fColType,`<br>`SQLCHAR *szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>`SQLCHAR *szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>`SQLCHAR *szTableName,`<br>`SQLSMALLINT cbTableName,`<br>`SQLUSMALLINT fScope,`<br>`SQLUSMALLINT fNullable );` | SQLRETURN **SQLSpecialColumnsW** (<br>`SQLHSTMT hstmt`<br>`SQLUSMALLINT fColType,`<br>`SQLWCHAR *szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>`SQLWCHAR *szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>`SQLWCHAR *szTableName,`<br>`SQLSMALLINT cbTableName,`<br>`SQLUSMALLINT fScope,`<br>`SQLUSMALLINT fNullable );` |

*Table 262. Comparison of suffix-W APIs to equivalent generic APIs (continued)*

| Generic APIs | Suffix-W APIs |
|---|---|
| SQLRETURN **SQLStatistics**(<br>`SQLHSTMT hstmt`<br>`SQLCHAR *szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>`SQLCHAR *szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>`SQLCHAR *szTableName,`<br>`SQLSMALLINT cbTableName,`<br>`SQLUSMALLINT fUnique,`<br>`SQLUSMALLINT fAccuracy );` | SQLRETURN **SQLStatisticsW** (<br>`SQLHSTMT hstmt`<br>**`SQLWCHAR`** `*szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>**`SQLWCHAR`** `*szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>**`SQLWCHAR`** `*szTableName,`<br>`SQLSMALLINT cbTableName,`<br>`SQLUSMALLINT fUnique,`<br>`SQLUSMALLINT fAccuracy );` |
| SQLRETURN **SQLTablePrivileges** (<br>`SQLHSTMT hstmt`<br>`SQLCHAR *szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>`SQLCHAR *szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>`SQLCHAR *szTableName,`<br>`SQLSMALLINT cbTableName );` | SQLRETURN **SQLTablePrivilegesW** (<br>`SQLHSTMT hstmt`<br>**`SQLWCHAR`** `*szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>**`SQLWCHAR`** `*szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>**`SQLWCHAR`** `*szTableName,`<br>`SQLSMALLINT cbTableName );` |
| SQLRETURN **SQLTables** (<br>`SQLHSTMT hstmt`<br>`SQLCHAR *szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>`SQLCHAR *szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>`SQLCHAR *szTableName,`<br>`SQLSMALLINT cbTableName,`<br>`SQLCHAR *szTableType,`<br>`SQLSMALLINT cbTableType );` | SQLRETURN **SQLTablesW** (<br>`SQLHSTMT hstmt`<br>**`SQLWCHAR`** `*szCatalogName,`<br>`SQLSMALLINT cbCatalogName,`<br>**`SQLWCHAR`** `*szSchemaName,`<br>`SQLSMALLINT cbSchemaName,`<br>**`SQLWCHAR`** `*szTableName,`<br>`SQLSMALLINT cbTableName,`<br>**`SQLWCHAR`** `*szTableType,`<br>`SQLSMALLINT cbTableType );` |

## Examples of handling the application encoding scheme

You can use the application encoding scheme to bind a UCS-2 result set column, bind UTF-8 data to parameter markers, retrieve UTF-8 data into application variables. The application encoding scheme also allows you to use suffix-W APIs. To perform these tasks, you must declare variables, specifying data types appropriately for a particular encoding scheme.

### Example of binding result set columns to retrieve UCS-2 data

You can use `SQLBindCol()` to bind the first column of a result set to a Unicode UCS-2 application buffer.

The following code shows such an example.

```
/* 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);
```

### Example of binding UTF-8 data to parameter markers

You can use `SQLBindParameter()` to bind application variables that contain UTF-8 data to INTEGER, CHAR, and GRAPHIC parameter markers.

The following example shows how `SQLBindParameter()` binds three input application variables containing UTF-8 to the 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 66. An application that binds application variables to parameter markers*

**Example of retrieving UTF-8 data into application variables**

You can use SQLGetData() to retrieve UTF-8 data from columns in the current row of a result set.

The following example shows how SQLGetData() can retrieve 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 67. An application that retrieves result set data into application variables*

**Example of using suffix-W APIs**

You can use suffix-W APIs to handle application encoding schemes.

The following example 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]
    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;
    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, ");
  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;
  /*****************************************************************/
  /*  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;
  /*****************************************************************/
  /*  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,
                        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,
                        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,
                        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;
 /* 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;
  /* 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
   */
  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 68. An application that uses suffix-W APIs*

## Embedded SQL and Db2 ODBC in the same program

You can combine embedded static SQL with Db2 ODBC to write a mixed application.For 64-bit applications, you cannot use embedded static SQL statements.

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.

The following example 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=
   /*  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=
   /*  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=
   /*  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=
   /*  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 69. An application that mixes embedded and dynamic SQL*

**Related concepts**

Rules for a Db2 ODBC stored procedure

Db2 ODBC stored procedures are like other Db2 ODBC applications. However, several differences exist.

# Vendor escape clauses

Vendor escape clauses increase the portability of your application if your application accesses multiple data sources from different vendors. However, if your application accesses only Db2 data sources, you have no reason to use vendor escape clauses.

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.

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.

**Related concepts**

ODBC-defined SQL extensions

ODBC supports various SQL extensions that are not supported by X/Open.

## Function for 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 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.

**Related concepts**

ODBC-defined SQL extensions

ODBC supports various SQL extensions that are not supported by X/Open.

## 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.

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.

**Related concepts**
ODBC-defined SQL extensions
ODBC supports various SQL extensions that are not supported by X/Open.

## ODBC-defined SQL extensions

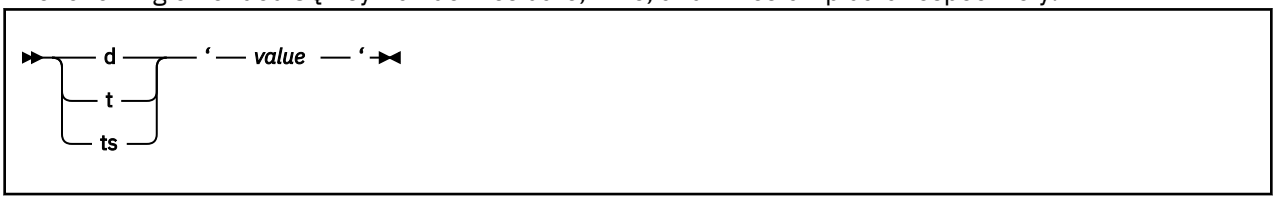
ODBC supports various SQL extensions that are not supported by X/Open.

The following list shows the SQL extensions that ODBC defines that 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

**ODBC date, time, and timestamp data**
You can use extended SQL syntax for date, time, and timestamp data in a vendor escape clause to make the definitions portable in your SQL statements.

The following extended SQL syntax defines date, time, and timestamp data respectively.




**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[.fffffffffff]* 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, you can use outer join syntax in a vendor escape clause to make outer joins portable in your SQL statements.

The following extended SQL syntax specifies an outer join.

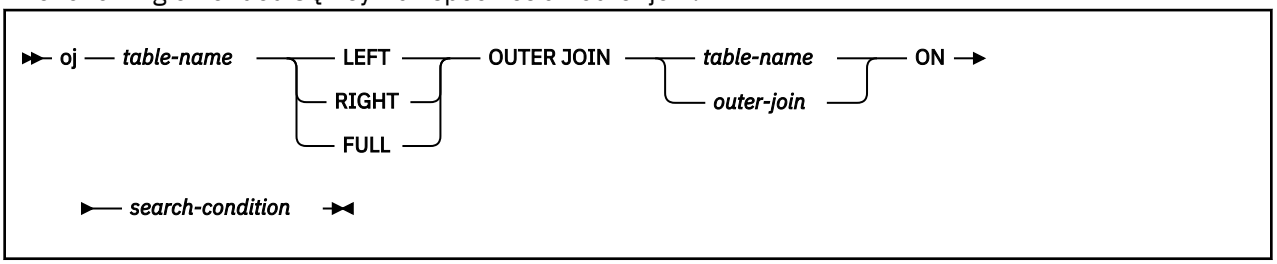



***table-name***
Specifies the name of the table that you want to join.

**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 '!'} '%!%'
```

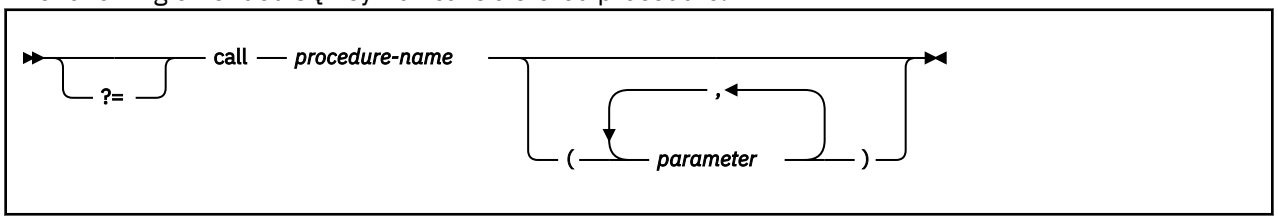
- Long-form syntax:

```
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, you can use the extended SQL syntax for calling a stored procedure in a vendor escape clause to make stored procedure calls portable in your SQL statements.

The following extended SQL syntax calls a stored procedure.




**?=**
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.

**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.

**Related concepts**

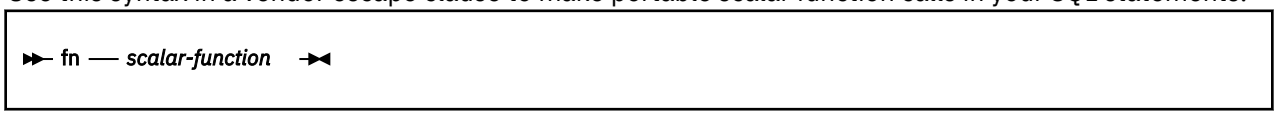
Stored procedures for ODBC applications

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.

**ODBC scalar functions**

You can use SQL scalar functions on columns of result sets, or on columns that restrict rows of a result set.

Use this syntax in a vendor escape clause to make portable scalar function calls in your SQL statements.




***scalar-function***
Specifies any string, date and time, and system functions.

**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

**Related reference**

Extended scalar functions

ODBC supports the use of extended scalar functions through vendor escape clauses. Each function can be called by using the escape clause syntax, or by calling the equivalent Db2 function.

# Extended indicators in ODBC applications

ODBC applications can use extended indicators to update all columns in UPDATE, INSERT, and MERGE statements without specifying the current value of columns that do not require changes.

If you use extended indicators you do not need to code separate INSERT statements for every combination of columns that you want to insert. You can enable extended indicators by setting the EXTENDEDINDICATOR keyword in the ODBC initialization file, or with the SQL_ATTR_EXTENDED_INDICATORS connection attribute. When you execute `SQLBindParameter()`, you can set the *rgbValue* to null and the *pcbValue* to SQL_DEFAULT_PARAM or SQL_UNASSIGNED.

You can also use the ODBC array input interface to bind a parameter marker to an array of application variables instead of a single application variable. You are not required to call SQLExecute() repeatedly on the same INSERT, UPDATE, or MERGE statement.

# ODBC programming hints and tips

When you program a Db2 ODBC application, you can avoid common problems, improve performance, reduce network flow, and maximize portability.

## Guidelines for avoiding common problems

To avoid common problems in Db2 ODBC initialization files, large result sets, and distinct types, adhere to the ODBC guidelines.

### Check the Db2 ODBC initialization file

You need to follow several guidelines to ensure that the Db2 ODBC initialization file is free from problems.

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.

### Limit the number of rows that an application can fetch

If a result set is too large, it might cause problems for the application. You can follow guidelines to reduce errors.

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.

### Cast parameter markers to distinct types or distinct types to source types

When you use a distinct-type parameter in the predicate of a query statement, you must use a CAST function. You can cast either the parameter marker to a distinct type, or you can 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 ) = ?
```

**Related reference**

CAST specification (Db2 SQL)

PREPARE (Db2 SQL)

## Techniques for improving application performance

You can follow several techniques to improve your 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.

### Set isolation levels for maximum concurrency and data consistency

Isolation levels determine the level of locking that is required to execute a statement and the level of concurrency that is possible in your application. 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 by statement, by connection, or at the driver level:

- `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.
- The TXNISOLATION initialization keyword sets the default isolation level for the Db2 ODBC driver.

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 database server.

**Related reference**

SQLSetConnectAttr() - Set connection attributes
`SQLSetConnectAttr()` sets attributes that govern aspects of connections.

SQLSetStmtAttr() - Set statement attributes
`SQLSetStmtAttr()` sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call `SQLSetConnectAttr()`.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

isolation-clause (Db2 SQL)

**Disable cursor hold behavior for more efficient resource use**

Db2 ODBC can more efficiently use resources that are 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.

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.

**Related reference**

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

**Code ODBC functions for efficient data retrieval**

You can retrieve data more efficiently by following several guidelines.

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.)

**Limit the use of catalog functions**

You can improve performance and reduce lock contention by limiting the use of catalog functions.

Limit the number of times that you call catalog functions in your application, limit the number of rows that the 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.

**Important:** Plan ahead. Although you might develop and test an application on a data source with hundreds of tables, the final application might need to run on a production database with thousands of tables.

### Enabling dynamic SQL statement caching for ODBC function calls

To reduce the overhead for function calls, you can prepare a statement once and execute it repeatedly throughout the application.

#### About this task

Db2 servers cache prepared versions of dynamic SQL statements. This dynamic caching allows the Db2 server to reuse previously prepared statements.

**Introductory concepts**

Submitting SQL statements to Db2 (Introduction to Db2 for z/OS)
Dynamic SQL applications (Introduction to Db2 for z/OS)

#### Procedure

To take advantage of dynamic caching for ODBC function calls, take any of the following actions:

- Use the same statement handle to execute identical SQL statements, and 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, 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.
- Set the LITERALREPLACEMENT property to 1 so that Db2 can share a cache entry for dynamic statements that are identical except for the literal constants and also meet the other standard criteria for sharing a cached entry.

  This sharing of the dynamic cache entry might improve your application performance.

**Related concepts**

Reoptimization for statements with replaced literal values (Db2 Performance)

**Related tasks**

Improving dynamic SQL performance (Db2 Performance)

**Related reference**

Db2 ODBC initialization keywords

Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

### Turn off statement scanning

To increase performance, you can configure Db2 ODBC to scan for vendor escape clauses only on handles that have escape clauses.

#### About this task

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.

#### Procedure

Set the SQL_ATTR_NOSCAN statement attribute to SQL_NOSCAN_ON.

You can set this attribute with either of the following functions:

| Option | Description |
|---|---|
| **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. |

#### Related concepts

Vendor escape clauses

Vendor escape clauses increase the portability of your application if your application accesses multiple data sources from different vendors. However, if your application accesses only Db2 data sources, you have no reason to use vendor escape clauses.

#### Related reference

SQLSetStmtAttr() - Set statement attributes

`SQLSetStmtAttr()` sets attributes that are related to a statement. To set an attribute for all statements that are associated with a specific connection, an application can call `SQLSetConnectAttr()`.

SQLSetConnectAttr() - Set connection attributes

`SQLSetConnectAttr()` sets attributes that govern aspects of connections.

## Techniques for reducing network flow

You can take several actions to reduce 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.

### Use `SQLSetColAttributes()` to reduce network flow

Whenever you prepare or execute a query statement directly, Db2 ODBC retrieves information about the SQL data type and size from the data source. If you use `SQLSetColAttributes()` to provide Db2 ODBC with this information in advance, Db2 ODBC does not need 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.

**Disable autocommit to reduce network flow**
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.

**Important:** SQL_AUTOCOMMIT_ON is the default setting for this attribute, unless it is otherwise specified in the initialization file.

**Related reference**
SQLSetConnectAttr() - Set connection attributes
SQLSetConnectAttr() sets attributes that govern aspects of connections.

Db2 ODBC initialization keywords
Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

**Use arrays to send and retrieve data**
For better performance, use arrays to update and retrieve data, and to pass parameter values and retrieve result sets.

Sending multiple data values through the network using arrays rather than individual application variables reduces network flow.

**Related concepts**
Using arrays to pass parameter values
Db2 ODBC provides an array input method for updating Db2 tables.

Retrieval of a result set into an array
An application can issue a query statement and fetch rows from the result set that the query generates.

**Use LOBs to manipulate 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.

## Techniques for maximizing application portability

You can maximize application portability by following several guidelines.

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().

**Use 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.

**Use 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

Several guidelines exist for working with the Db2 ODBC traces, including information about general diagnosis, debugging, and abnormal terminations.

You can obtain traces for Db2 ODBC applications and diagnostics and Db2 ODBC stored procedures.

## ODBC trace types

Db2 ODBC provides two traces that differ in purpose: an application trace for debugging user applications, and a service trace for problem diagnosis.

### Application trace

You can enable Db2 ODBC application trace by 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 designed to trace ODBC API calls. The trace is written to the file specified on the APPLTRACEFILENAME keyword.

Use this trace to debug your Db2 ODBC applications.

#### Formats for the trace file name

To specify the APPLTRACEFILENAME keyword setting, you can use either JCL DD statement format or z/OS UNIX environment HFS format.

##### JCL DD statement format for a trace file name

The primary use of the JCL DD 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 JCL DD statement format is `APPLTRACEFILENAME="DD:ddname"`. The *ddname* value is the name of the DD statement that is specified in your job or TSO logon procedure.

**Example:** Assume that the keyword setting is `APPLTRACEFILENAME="DD:APPLDD"`. You can use the following JCL DD 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 for a trace file name

The z/OS UNIX environment HFS format is used only for writing to HFS files.

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.

© Copyright IBM Corp. 1997, 2020

**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
```

**Example of application trace output**

The Db2 ODBC trace facility records information, including the APIs that are started, values that are used, and data pointers.

The following is an application trace output of the same multi-threaded application using TRACEPIDTID=1. The execution path of the SQLAllocHandle() call can be easily followed using the PID and TID information.

- Line 5, 10, and 11 - Call to SQLAllocHandle to allocate a connection handle ( phOutput=2 )
- Line 12, 19, and 20 - Call to SQLConnect to establish a connection to the target data source
- Line 21, 29, and 30 - Call to SQLAllocHandle to allocate a statement handle ( phOutput=2 )
- Line 32, 54, and 55 - Call to SQLExecDirect to execute a SELECT on the statement handle
- Line 56, 57, and 58 - Call to SQLEndTran to commit the transaction
- Line 59, 68, and 69 - Call to SQLFreeHandle to free the statement handle
- Line 70, 71, and 72 - Call to SQLDisconnect to disconnect from the target data source
- Line 73, 74, and 75 - Call SQLFreeHandle to free the connection handle

```
1  [0400000D 00000000184B2000] SQLAllocHandle( fHandleType=SQL_HANDLE_ENV, hInput=0, .. )
2  [0400000D 00000000184B2000]
SQLAllocHandle( phOutput=1 )
3  [0400000D 00000000184B2000]     --->
SQL_SUCCESS

4  [0400000D 00000000184B5000] SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=1, .. )
5  [0400000D 00000000184B4000] SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=1, .. )
6  [0400000D 00000000184B3000] SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=1, .. )
7  [0400000D 00000000184B5000]
SQLAllocHandle( phOutput=1 )
8  [0400000D 00000000184B5000]     ---> SQL_SUCCESS

9  [0400000D 00000000184B5000] SQLConnect( hDbc=1, szDSN="STLEC1", cbDSN=-3, szUID=Null
Pointer, .. )
10 [0400000D 00000000184B4000}
SQLAllocHandle( phOutput=2 )
11 [0400000D 00000000184B4000]     --->
SQL_SUCCESS

12 [0400000D 00000000184B4000] SQLConnect( hDbc=2, szDSN="STLEC1", cbDSN=-3, szUID=Null
Pointer, .. )
13 [0400000D 00000000184B3000]
SQLAllocHandle( phOutput=3 )
14 [0400000D 00000000184B3000]     --->
SQL_SUCCESS

15 [0400000D 00000000184B3000] SQLConnect( hDbc=3, szDSN="STLEC1", cbDSN=-3, szUID=Null
Pointer, .. )
16 [0400000D 00000000184B5000]
SQLConnect( )
17 [0400000D 00000000184B5000]     ---> SQL_SUCCESS

18 [0400000D 00000000184B5000] SQLAllocHandle( fHandleType=SQL_HANDLE_STMT, hInput=1, .. )
19 [0400000D 00000000184B4000]
SQLConnect( )
20 [0400000D 00000000184B4000]     --->
SQL_SUCCESS

21 [0400000D 00000000184B4000] SQLAllocHandle( fHandleType=SQL_HANDLE_STMT, hInput=2, .. )
22 [0400000D 00000000184B3000]
SQLConnect( )
23 [0400000D 00000000184B3000]     --->
SQL_SUCCESS

24 [0400000D 00000000184B3000] SQLAllocHandle( fHandleType=SQL_HANDLE_STMT, hInput=3, .. )
25 [0400000D 00000000184B5000]
SQLAllocHandle( phOutput=1 )
26 [0400000D 00000000184B5000]     --->
SQL_SUCCESS

27 [0400000D 00000000184B3000]
SQLAllocHandle( phOutput=3 )
28 [0400000D 00000000184B3000]     --->
SQL_SUCCESS

29 [0400000D 00000000184B4000]
SQLAllocHandle( phOutput=2 )
30 [0400000D 00000000184B4000]     ---> SQL_SUCCESS

31 [0400000D 00000000184B5000] SQLExecDirect( hStmt=1, pszSqlStr="SELECT * FROM
SYSIBM.SYSDUMMY1", ..)
32 [0400000D 00000000184B4000] SQLExecDirect( hStmt=2, pszSqlStr="SELECT * FROM
SYSIBM.SYSDUMMY1", ..)
33 [0400000D 00000000184B3000] SQLExecDirect( hStmt=3, pszSqlStr="SELECT * FROM
SYSIBM.SYSDUMMY1", ..)
34 [0400000D 00000000184B5000]
SQLExecDirect( )
35 [0400000D 00000000184B5000]     ---> SQL_SUCCESS

36 [0400000D 00000000184B5000} SQLEndTran( fHandleType=SQL_HANDLE_DBC, hHandle=1, .. )
37 [0400000D 00000000184B5000]
SQLEndTran( )
38 [0400000D 00000000184B5000]     --->
SQL_SUCCESS

39 [0400000D 00000000184B3000]
SQLExecDirect( )
40 [0400000D 00000000184B3000]     --->
SQL_SUCCESS

41 [0400000D 00000000184B3000] SQLEndTran( fHandleType=SQL_HANDLE_DBC, hHandle=3, .. )
42 [0400000D 00000000184B3000]
SQLEndTran( )
43 [0400000D 00000000184B3000]     --->
SQL_SUCCESS
```

**Related concepts**

Db2 ODBC initialization file

A set of optional keywords can be specified in a Db2 ODBC *initialization file*. An initialization file stores default values for various Db2 ODBC configuration options. Because the initialization file has EBCDIC text, you can use a file editor, such as the TSO editor, to edit it.

## ODBC diagnostic trace

The diagnostic trace captures information to use in Db2 ODBC problem determination. Use this trace only under the direction of IBM Support. This trace is not intended to assist in debugging user-written Db2 ODBC applications.

You can view the diagnostic 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.

**Related reference**

Db2 ODBC initialization keywords

Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

### Capturing ODBC diagnostic trace information in z/OS

Use the ODBC diagnostic trace only under the direction of IBM Support.

**Procedure**

To capture ODBC diagnostic trace information in z/OS:

1. Activate the diagnostic trace by performing one of the following actions:
   - Specify DIAGTRACE=1 in the Db2 ODBC initialization file.

     If you activate the diagnostic trace by using the DIAGTRACE keyword in the initialization file, you must also allocate a DSNAOTRC data set in your job or TSO logon procedure. You can use one of the following methods to allocate a DSNAOTRC data set:

     – Specify a DSNAOTRC DD statement in your job or TSO logon procedure.
     – Use the TSO/E ALLOCATE command.
     – Use dynamic allocation in your ODBC application.
   - Issue one of the following ODBC diagnostic trace commands:
     – DSNAOTRC command for non-XPLINK applications
     – DSNAOTRX command for 31-bit XPLINK applications
     – DSNAO64T command for 64-bit applications

   **Recommendation:** When tracing 64-bit applications, consider specifying a diagnostic trace buffer size that is up to 10% bigger than what you would specify for 31-bit applications. If you activated the trace by specifying DIAGTRACE=1 in the ODBC initialization file, specify this buffer size with the DIAGTRACE_BUFFER_SIZE keyword in the initialization file. If you activated the trace by using a diagnostic trace command, specify this buffer size as a parameter of that command.
2. Stop, dump, or format the trace as needed by using the appropriate ODBC diagnostic trace command with the relevant options or by changing the values of the appropriate keywords in the ODBC initialization file.

**Related concepts**

Specifications for the diagnostic trace file

The diagnostic trace data can be written to a z/OS sequential data set or to a z/OS UNIX environment HFS file.

**Related reference**

Db2 ODBC initialization keywords

Db2 ODBC initialization keywords control the run time environment for Db2 ODBC applications.

ODBC diagnostic trace commands

Use an ODBC diagnostic trace command to start and stop traces, query the status of a trace, capture the trace table, and format the trace information. Use the ODBC diagnostic trace only under the direction of IBM Software Support.

**Capturing ODBC diagnostic trace information in the z/OS UNIX environment**

Use the ODBC diagnostic trace only under the direction of IBM Support. You can activate this diagnostic trace from the z/OS UNIX environment command line.

**Procedure**

To capture ODBC diagnostic trace information in the z/OS UNIX environment:

1. Use the TSO/E command, OPUTX, to store the following program load modules in z/OS UNIX environment HFS files:
   - DSNAOTRC, for non-XPLINK applications
   - DSNAOTRX, for 31-bit XPLINK applications
   - DSNAO64T, for 64-bit applications

   The following example uses the OPUTX command to store load module DSNAOTRC from the partitioned data set *prefix*.SDSNLOAD to the HFS file DSNAOTRC in the directory /usr/db2:

   ```
   OPUTX  'prefix.SDSNLOAD(DSNAOTRC)'  /usr/db2/dsnaotrc
   ```

   The following example uses the OPUTX command to store load module DSNAOTRX from the partitioned data set *prefix*.SDSNLOD2 to the HFS file DSNAOTRX in the directory /usr/db2:

   ```
   OPUTX  'prefix.SDSNLOD2(DSNAOTRX)'  /usr/db2/dsnaotrx
   ```

   The following example uses the OPUTX command to store load module DSNAO64T from the partitioned data set *prefix*.SDSNLOD2 to the HFS file DSNAO64T in the directory /usr/db2:

   ```
   OPUTX  'prefix.SDSNLOD2(DSNAO64T)'  /usr/db2/dsnao64t
   ```

2. 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.
3. Go to the directory that contains the DSNAOTRC, DSNAOTRX, and DSNAO64T load modules.
4. Verify that execute permission is established for the DSNAOTRC, DSNAOTRX, and DSNAO64T load modules. If execute permission was not granted, use the chmod command to set execute permission for each of the load modules.
5. Issue one of the following ODBC diagnostic trace commands from the z/OS UNIX environment command line to activate the diagnostic trace:

| Option | Description |
|---|---|
| **dsnaotrc on** | Use this command when tracing non-XPLINK applications. |
| **dsnaotrx on** | Use this command when tracing 31-bit XPLINK applications. |
| **dsnao64t on** | Use this command when tracing 64-bit XPLINK applications. |

   The options for activating the diagnostic trace are optional.
6. Run the ODBC application.

7. Issue one of the following ODBC diagnostic trace commands from the z/OS UNIX environment command line to dump the diagnostic trace:

| Option | Description |
|---|---|
| **dsnaotrc dmp "*raw_trace_file*"** | Use this command when tracing non-XPLINK applications. |
| **dsnaotrx dmp "*raw_trace_file*"** | Use this command when tracing 31-bit XPLINK applications. |
| **dsnao64t dmp "*raw_trace_file*"** | Use this command when tracing 64-bit XPLINK applications. |

   The *raw_trace_file* value is the name of the output file to which Db2 writes the raw diagnostic trace data.

   Each of the following example statements show how to code the trace data set specification in the non-XPLINK environment. To code the following data set specifications in the XPLINK environment, replace the DSNAOTRC command with the DSNAOTRX or DSNAO64 command. All additional syntax for the trace command in each environment is identical.

   - Currently allocated JCL DD 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"
     ```

8. Issue one of the following ODBC diagnostic trace commands from the z/OS UNIX environment command line to deactivate the diagnostic trace:

| Option | Description |
|---|---|
| **dsnaotrc off** | Use this command when tracing non-XPLINK applications. |
| **dsnaotrx off** | Use this command when tracing 31-bit XPLINK applications. |
| **dsnao64t off** | Use this command when tracing 64-bit XPLINK applications. |

9. Issue one of the following ODBC diagnostic trace commands from the z/OS UNIX environment command line to format the raw trace data records from input file *raw_trace_file* to output file *fmt_trace_file*.

| Option | Description |
|---|---|
| **dsnaotrc fmt "*raw_trace_file*" "*fmt_trace_file*"** | Use this command when requesting detailed trace reports for non-XPLINK applications. |
| **dsnaotrc flw "*raw_trace_file*" "*fmt_trace_file*"** | Use this command when requesting flow trace reports for non-XPLINK applications. |
| **dsnaotrx fmt "*raw_trace_file*" "*fmt_trace_file*"** | Use this command when requesting detailed trace reports for 31-bit XPLINK applications. |
| **dsnaotrx flw "*raw_trace_file*" "*fmt_trace_file*"** | Use this command when requesting flow trace reports for 31-bit XPLINK applications. |

| Option | Description |
|---|---|
| **dsnao64t fmt "*raw_trace_file*" "*fmt_trace_file*"** | Use this command when requesting detailed trace reports for 64-bit XPLINK applications. |
| **dsnao64t flw "*raw_trace_file*" "*fmt_trace_file*"** | Use this command when requesting flow trace reports for 64-bit XPLINK applications. |

Each of the following statements show how to code the input data set specification in the non-XPLINK environment. To code the following data set specifications in the XPLINK environment, replace the DSNAOTRC command with the DSNAOTRX or DSNAO64 command. All additional syntax for the trace command in each environment is identical.

- Currently allocated JCL DD 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
```

Each of the following example statements show how to code the output data set specification in the non-XPLINK environment. To code the following data set specifications in the XPLINK environment, replace the DSNAOTRC command with the DSNAOTRX or DSNAO64 command. All additional syntax for the trace command in each environment is identical.

- Currently allocated JCL DD 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"
```

10. Delete the DSNAOTRC, DSNAOTRX, and DSNAO64T program modules from your z/OS UNIX environment directory. Do not attempt to maintain a private copy of the these program modules in your HFS directory.

**Related concepts**

Specifications for the diagnostic trace file

The diagnostic trace data can be written to a z/OS sequential data set or to a z/OS UNIX environment HFS file.

**Related reference**

ODBC diagnostic trace commands

Use an ODBC diagnostic trace command to start and stop traces, query the status of a trace, capture the trace table, and format the trace information. Use the ODBC diagnostic trace only under the direction of IBM Software Support.

### ODBC diagnostic trace commands

Use an ODBC diagnostic trace command to start and stop traces, query the status of a trace, capture the trace table, and format the trace information. Use the ODBC diagnostic trace only under the direction of IBM Software Support.

The following trace commands perform the same tracing tasks and use identical syntax:

**DSNAOTRC**
Use this trace command for non-XPLINK ODBC applications.

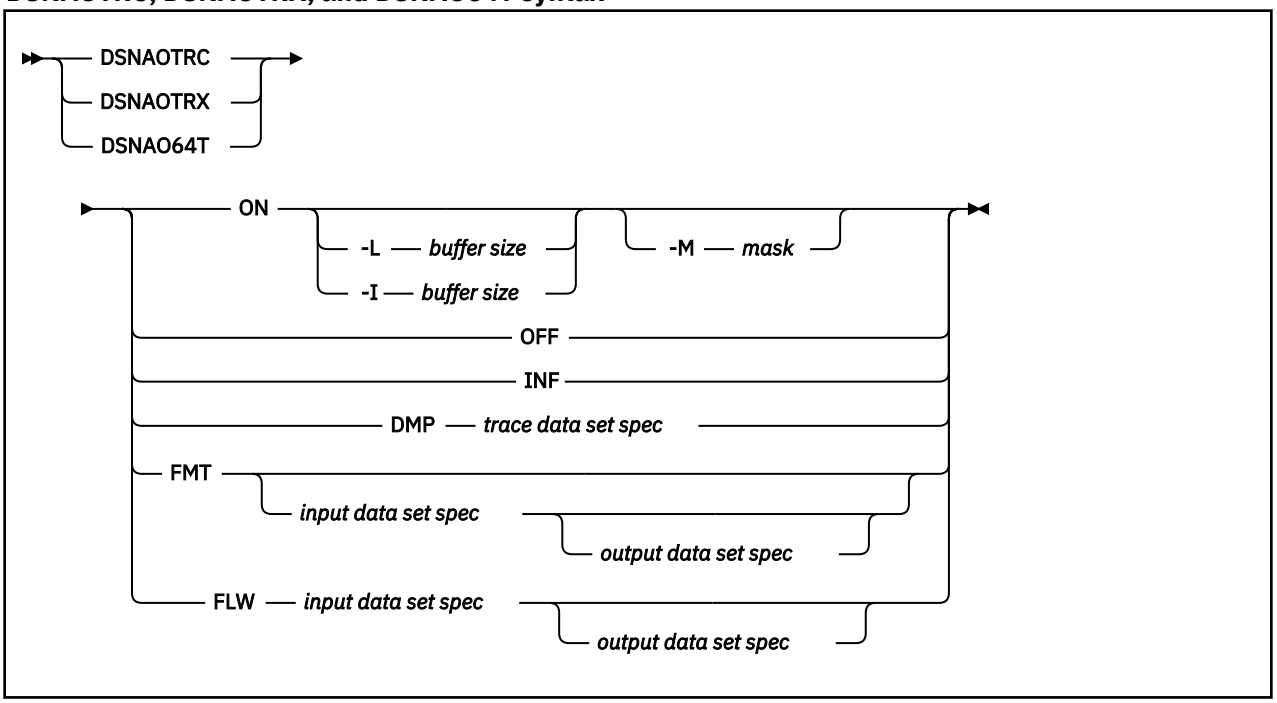
**DSNAOTRX**
Use this trace command for 31-bit XPLINK ODBC applications.

**DSNAO64T**
Use this trace command for 64-bit ODBC applications.

These trace commands 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.

### DSNAOTRC, DSNAOTRX, and DSNAO64T syntax




### DSNAOTRC, DSNAOTRX, and DSNAO64T 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 DD statement name.

**FMT**
Generate a formatted detail report of the Db2 ODBC diagnostic trace contents.

Trace data captured through DSNAO64T must be formatted with the DSNAO64T command. Otherwise, trace data can become corrupted.

**FLW**
Generate a formatted flow report of the Db2 ODBC diagnostic trace contents.

***input data set spec***
DSNAO64T The data set that contains the raw Db2 ODBC diagnostic trace data to be formatted. This data set was generated as the result of a DSNAOTRC DMP command, the DSNAOTRX DMP command, or the DSNAO64T DMP command or by the DSNAOTRC DD 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 DD statement name. If this parameter is not specified, the DSNAOTRC, DSNAOTRX, or DSNAO64T 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 DD 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").

**Related concepts**
Extra performance linkage
The XPLINK Db2 ODBC driver enhances the performance of XPLINK ODBC applications. The XPLINK Db2 ODBC driver is recommended to enhance performance only if your ODBC application uses XPLINK code exclusively.

**Specifications for the diagnostic trace file**
The diagnostic trace data can be written to a z/OS sequential data set or to 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 DD statement is not available to the ODBC application. You can format DD:DSNAOTRC by using the DSNAOTRC, DSNAOTRX, or DSNAO64T trace formatting programs.

**Example:** The following JCL examples use a DSNAOTRC DD 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)
```

## Stored procedure trace

To obtain an application trace or a diagnostic trace of a Db2 ODBC stored procedure, you need to perform certain steps in the initialization file.

Db2 ODBC stored procedures run in 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 space 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 procedure address space that uses the Db2 ODBC application trace or diagnostic trace.

### Tracing a client application

Tracing a client application requires setting certain parameters in the common section of the initialization file. Additionally, you must specify an APPLTRC DD statement in the JCL for the application job or in your TSO logon procedure.

#### Procedure

To obtain an application trace:

1. Set APPLTRACE=1 and APPLTRACEFILENAME="DD:*dd-name*" in the common section of the Db2 ODBC initialization file as follows:

   ```
   [COMMON]
   APPLTRACE=1
   APPLTRACEFILENAME="DD:APPLTRC"
   ```

   *dd-name* is the name of a DD statement specified in the JCL for the application job or your TSO logon procedure.
2. Specify an APPLTRC DD statement in the JCL for the application job or your TSO logon procedure.

   The DD 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'
   ```

**Related reference**

SQL to C data conversion
To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

**Obtaining an application trace for a stored procedure**

Obtaining an application trace for a stored procedure requires modifying the initialization file.

**Procedure**

To obtain an application trace:

1. Set APPLTRACE=1 and APPLTRACEFILENAME="DD:*dd-name*" in the common section of the Db2 ODBC initialization file as follows:

   ```
   [COMMON]
   APPLTRACE=1
   APPLTRACEFILENAME="DD:APPLTRC"
   ```

   *dd-name* is the name of a DD statement that is specified in the JCL for the stored procedure address space.
2. Specify an JCL DD statement in the JCL for the stored procedure address space.

   The DD 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'
   ```

**Obtaining a diagnostic trace for a stored procedure**

You can trace a stored procedure with the ODBC diagnostic trace if the stored procedure is an ODBC application program. Use the ODBC diagnostic trace only under the direction of IBM Support.

**Procedure**

To obtain a diagnostic trace for a stored procedure:

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. *nnnnnnn* is the number of bytes to allocate for the diagnostic trace buffer.
   For example:

   ```
   [COMMON]
   DIAGTRACE=1
   DIAGTRACE_BUFFER_SIZE=2000000
   DIAGTRACE_NO_WRAP=1
   ```

   **Recommendation:** When tracing 64-bit applications, consider specifying a diagnostic trace buffer size that is up to 10% bigger than what you would specify for 31-bit applications.
2. Specify a z/OS DSNAOINI DD statement in the JCL for the stored procedure address space. The DD 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 DD statement in the JCL for the stored procedures space. The DD 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 procedure address space by using the z/OS command, "VARY WLM,APPLENV=*name*,QUIESCE". *name* is the WLM application environment name.
6. Submit either the formatted or unformatted diagnostic trace data to IBM Support. To format the raw trace data at your site, run the DSNAOTRC command (or the DSNAOTRX command in the XPLINK environment or the DSNAO64T command in the 64-bit environment) with the FMT or FLW options against the diagnostic trace data set.

**Obtaining an application trace for a client application and a stored procedure**
You can debug your stored procedure and the calling application with an ODBC application trace if both the application and stored procedure are Db2 ODBC application programs.

**Procedure**

To obtain an application trace for a client application and a stored procedure:

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 DD statement specified in both the JCL for the client application job and the stored procedure address space.
2. Specify a APPLTRC DD statement in the JCL for the client application. The DD 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 DD statement in the JCL for the stored procedure address space. The DD 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.

**Obtaining a diagnostic trace for a client application and a stored procedure**
You can trace a stored procedure and its calling application with the ODBC diagnostic trace if both the application and stored procedure are ODBC application programs. Use the ODBC diagnostic trace only under the direction of IBM Software Support.

**Procedure**

To obtain a diagnostic trace for a client application and a stored procedure:

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. *nnnnnnn* is the number of bytes to allocate for the diagnostic trace buffer.
   For example:

   ```
   [COMMON]
   DIAGTRACE=1
   DIAGTRACE_BUFFER_SIZE=2000000
   DIAGTRACE_NO_WRAP=1
   ```

   **Recommendation:** When tracing 64-bit applications, consider specifying a diagnostic trace buffer size that is up to 10% bigger than what you would specify for 31-bit applications.
2. Specify a z/OS DSNAOINI DD statement in the JCL for the stored procedure address space. The DD 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 DD statement in JCL for the client application job. The DD 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 DD statement in the JCL for the stored procedure address space. The DD 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 procedure address space by using the following z/OS command:

   ```
   VARY WLM,APPLENV=name,QUIESCE
   ```

   *name* is the WLM application environment name.
7. Submit either the formatted or unformatted diagnostic trace data to IBM Support. To format the raw trace data at your site, run the DSNAOTRC command (or the DSNAOTRX command in the XPLINK environment or the DSNAO64T command in the 64-bit environment) with the FMT or FLW options against the client application's diagnostic trace data set and the stored procedure's diagnostic trace data set.

## 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. Database management systems 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.

**Related reference**

Language Environment Debugging Guide (z/OS Language Environment Debugging 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 useful 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
```

# Chapter 7. Db2 ODBC reference information

To use Db2 ODBC effectively, you need to know about differences between ODBC and Db2 ODBC, Db2 ODBC data types, extended scalar functions, SQLSTATEs, deprecated functions. You must also know where to locate example code.

## Db2 ODBC and ODBC differences

Db2 ODBC and standard ODBC differ in several ways for their drivers, data types, and isolation levels.

**Related reference**

Status of support for ODBC functions
Each function has its own ODBC 3.0 conformance level, and Db2 ODBC support level, and certain functions are deprecated.

### Db2 ODBC and ODBC drivers

Differences exist between the Db2 ODBC and ODBC drivers. Generally, Db2 ODBC supports a subset of the functions that the ODBC driver provides.

The following figure 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.

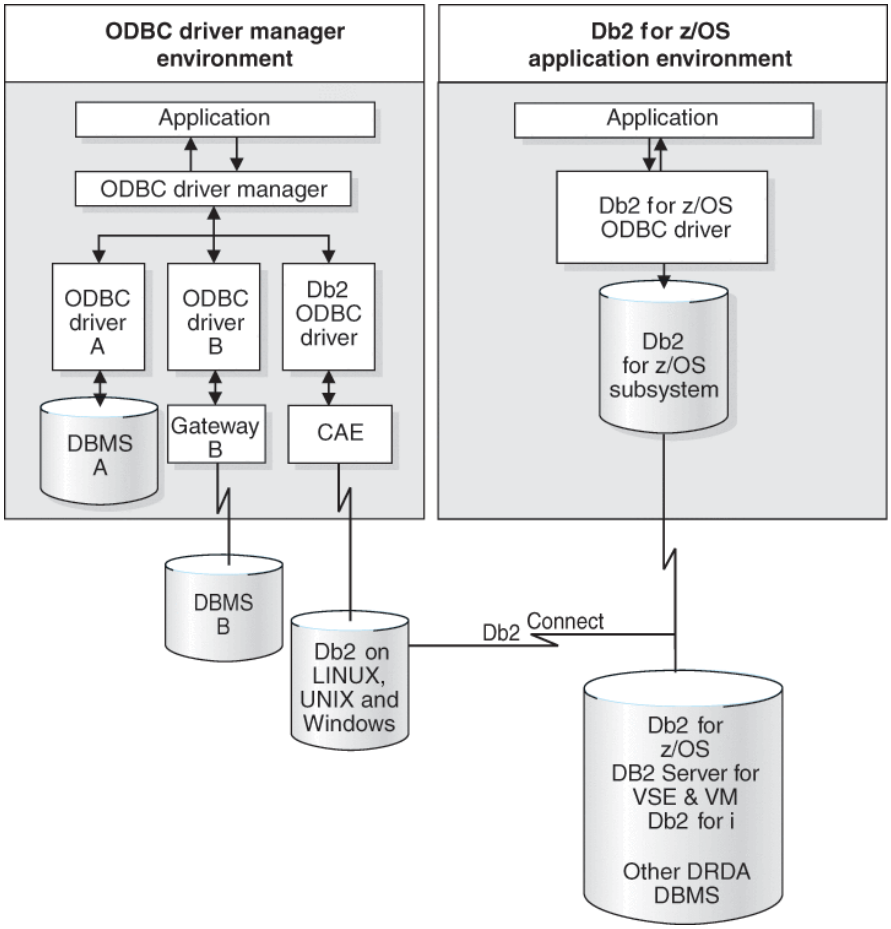



*Figure 71. Db2 ODBC and ODBC*

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

© Copyright IBM Corp. 1997, 2020

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 2.0specifications.

## ODBC APIs and data types

Db2 ODBC supports a subset of the functions that the ODBC driver provides.

The following table summarizes the ODBC 3.0 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 263. Db2 ODBC support*

| ODBC features | Db2 ODBC |
|---|---|
| Core level functions | All, except for:<br>• SQLDrivers()<br>• SQLGetDescField()<br>• SQLSetDescField()<br>• SQLGetDescRec()<br>• SQLSetDescRec()<br>• SQLCopyDesc() |
| Level 1 functions | All, except for `SQLBrowseConnect()`. |
| Level 2 functions | All |
| Additional Db2 ODBC functions | • SQLSetConnection()<br>• SQLGetEnvAttr()<br>• SQLSetColAttributes()<br>• SQLGetLength()<br>• SQLGetPosition()<br>• SQLGetSubString()<br>• SQLGetSQLCA() |
| Minimum SQL data types | • SQL_CHAR<br>• SQL_LONGVARCHAR<br>• SQL_VARCHAR |
| Core SQL data types | • SQL_DECIMAL<br>• SQL_NUMERIC<br>• SQL_SMALLINT<br>• SQL_INTEGER<br>• SQL_REAL<br>• SQL_FLOAT<br>• SQL_DOUBLE |

*Table 263. Db2 ODBC support (continued)*

| ODBC features | Db2 ODBC |
|---|---|
| Extended SQL data types | • SQL_BIT<br>• SQL_TINYINT<br>• SQL_BIGINT<br>• SQL_BINARY<br>• SQL_BLOB<br>• SQL_BLOB_LOCATOR<br>• SQL_CLOB<br>• SQL_CLOB_LOCATOR<br>• SQL_DBCLOB<br>• SQL_DBCLOB_LOCATOR<br>• SQL_DECFLOAT<br>• SQL_LONGVARBINARY<br>• SQL_ROWID<br>• SQL_TYPE_DATE<br>• SQL_TYPE_TIME<br>• SQL_TYPE_TIMESTAMP<br>• SQL_VARBINARY<br>• SQL_XML |
| ODBC 3.0 SQL data types | • SQL_GRAPHIC<br>• SQL_LONGVARGRAPHIC<br>• SQL_VARGRAPHIC |
| Core C data types | • SQL_C_CHAR<br>• SQL_C_DOUBLE<br>• SQL_C_FLOAT<br>• SQL_C_LONG (SLONG, ULONG)<br>• SQL_C_SHORT (SSHORT, USHORT) |
| Extended C data types | • SQL_C_BINARY<br>• SQL_C_BIT<br>• SQL_C_BLOB_LOCATOR<br>• SQL_C_CLOB_LOCATOR<br>• SQL_C_DBCLOB_LOCATOR<br>• SQL_C_DECIMAL64<br>• SQL_C_DECIMAL128<br>• SQL_C_TYPE_DATE<br>• SQL_C_TYPE_TIME<br>• SQL_C_TYPE_TIMESTAMP<br>• SQL_C_TINYINT |
| ODBC 3.0 C data types | • SQL_C_DBCHAR |

*Table 263. Db2 ODBC support (continued)*

| ODBC features | Db2 ODBC |
|---|---|
| Return codes | • SQL_SUCCESS<br>• SQL_SUCCESS_WITH_INFO<br>• SQL_NEED_DATA<br>• SQL_NO_DATA_FOUND<br>• SQL_ERROR<br>• 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()`. |

**Related reference**

Status of support for ODBC functions
Each function has its own ODBC 3.0 conformance level, and Db2 ODBC support level, and certain functions are deprecated.

**Related information**

Microsoft open database connectivity (ODBC)

## Isolation levels

With the exception of the no commit isolation level, all Db2 isolation levels have corresponding ODBC isolation levels.

The following table maps Db2 isolation levels to ODBC transaction isolation levels. The `SQLGetInfo()` function indicates which isolation levels are available.

*Table 264. Isolation levels under ODBC*

| Db2 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.

# Extended scalar functions

ODBC supports the use of extended scalar functions through vendor escape clauses. Each function can be called by using the escape clause syntax, or by calling the equivalent Db2 function.

**Related concepts**

ODBC scalar functions

You can use SQL scalar functions on columns of result sets, or on columns that restrict rows of a result set.

## Errors returned by extended scalar functions

All errors that are detected by extended scalar functions return SQLSTATE 38552 when you are connected to a Db2 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

Db2 supports various string functions that are 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

Db2 supports date and time functions that are defined through 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

Db2 supports system functions that are defined through vendor escape clauses.

The following rules apply to the arguments in these system functions:

- 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.)

## SQLSTATE cross reference

SQLSTATEs are returned by the Db2 ODBC application as a diagnostic tool for encountered errors, indicating the cause for these errors.

Table 265 on page 527 is a cross-reference of all the SQLSTATEs for each function. 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.

**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 265. SQLSTATE cross reference*

| SQLSTATE | Description | Functions |
|---|---|---|
| **01**000 | Warning. | • SQLAllocHandle()<br>• SQLCloseCursor()<br>• SQLColAttribute()<br>• SQLDescribeParam()<br>• SQLEndTran()<br>• SQLFreeHandle()<br>• SQLGetConnectAttr()<br>• SQLGetStmtAttr()<br>• SQLSetConnectAttr()<br>• SQLSetStmtAttr() |
| **01**002 | Disconnect error. | • SQLDisconnect() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **01**004 | Data truncated. | • SQLColAttribute()<br>• SQLDataSources()<br>• SQLDescribeCol()<br>• SQLDriverConnect()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLGetConnectAttr()<br>• SQLGetCursorName()<br>• SQLGetData()<br>• SQLGetDiagRec()<br>• SQLGetInfo()<br>• SQLGetStmtAttr()<br>• SQLGetSubString()<br>• SQLNativeSql()<br>• SQLPutData()<br>• SQLSetColAttributes() |
| **01**504 | The UPDATE or DELETE statement does not include a WHERE clause. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **01**S00 | Invalid connection string attribute. | • SQLDriverConnect() |
| **01**S01 | Error in row. | • SQLExtendedFetch() |
| **01**S02 | Option value changed. | • SQLDriverConnect()<br>• SQLSetConnectAttr()<br>• SQLSetStmtAttr() |
| **07**001 | Wrong number of parameters. | • SQLExecDirect()<br>• SQLExecute() |
| **07**002 | Too many columns. | • SQLExtendedFetch()<br>• SQLFetch() |
| **07**005 | The statement did not return a result set. | • SQLColAttribute()<br>• SQLDescribeCol() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **07**006 | Invalid conversion. | • SQLBindParameter()<br>• SQLExecDirect()<br>• SQLExecute()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLGetData()<br>• SQLGetLength()<br>• SQLGetPosition()<br>• SQLGetSubString() |
| **08**001 | Unable to connect to data source. | • SQLConnect() |
| **08**002 | Connection in use. | • SQLConnect() |
| **08**003 | Connection is closed. | • SQLAllocHandle()<br>• SQLDisconnect()<br>• SQLEndTran()<br>• SQLFreeHandle()<br>• SQLGetConnectAttr()<br>• SQLGetInfo()<br>• SQLNativeSql()<br>• SQLSetConnectAttr()<br>• SQLSetConnection() |
| **08**004 | The application server rejected establishment of the connection. | • SQLConnect() |
| **08**007 | Connection failure during transaction. | • SQLEndTran() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **08**S01 | Communication link failure. | • SQLBindCol()<br>• SQLBindFileToCol()<br>• SQLBindParameter()<br>• SQLBindFileToParameter()<br>• SQLCancel()<br>• SQLColumnPrivileges()<br>• SQLColumns()<br>• SQLDescribeCol()<br>• SQLExecDirect()<br>• SQLExecute()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLForeignKeys()<br>• SQLFreeStmt()<br>• SQLGetCursorName()<br>• SQLGetData()<br>• SQLGetFunctions()<br>• SQLGetInfo()<br>• SQLGetLength()<br>• SQLGetPosition()<br>• SQLGetSubString()<br>• SQLGetTypeInfo()<br>• SQLMoreResults()<br>• SQLNumParams()<br>• SQLNumResultCols()<br>• SQLParamData()<br>• SQLParamOptions()<br>• SQLPrepare()<br>• SQLPrimaryKeys()<br>• SQLProcedureColumns()<br>• SQLProcedures()<br>• SQLPutData()<br>• SQLRowCount()<br>• SQLSetColAttributes()<br>• SQLSetCursorName()<br>• SQLSetConnectAttr()<br>• SQLSetStmtAttr()<br>• SQLSpecialColumns()<br>• SQLStatistics()<br>• SQLTablePrivileges()<br>• SQLTables() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **0F**001 | The LOB token variable does not currently represent any value. | • SQLGetLength()<br>• SQLGetPosition()<br>• SQLGetSubString() |
| **21**S01 | Insert value list does not match column list. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **21**S02 | Degrees of derived table does not match column list. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **22**001 | String data right truncation. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPutData() |
| **22**002 | Invalid output or indicator buffer specified. | • SQLExtendedFetch()<br>• SQLFetch()<br>• SQLGetData() |
| **22**008 | Invalid datetime format or datetime field overflow. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLParamData()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLGetData()<br>• SQLPutData() |
| **22**011 | A substring error occurred. | • SQLGetSubString() |
| **22**012 | Division by zero is invalid. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLExtendedFetch()<br>• SQLFetch() |
| **22**018 | Error in assignment. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLGetData()<br>• SQLPutData() |
| **23**000 | Integrity constraint violation. | • SQLExecDirect()<br>• SQLExecute() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **24**000 | Invalid cursor state. | • SQLCloseCursor()<br>• SQLColumnPrivileges()<br>• SQLColumns()<br>• SQLExecDirect()<br>• SQLExecute()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLForeignKeys()<br>• SQLGetData()<br>• SQLGetStmtAttr()<br>• SQLGetTypeInfo()<br>• SQLPrepare()<br>• SQLPrimaryKeys()<br>• SQLProcedureColumns()<br>• SQLProcedures()<br>• SQLSetColAttributes()<br>• SQLSetStmtAttr()<br>• SQLSpecialColumns()<br>• SQLStatistics()<br>• SQLTablePrivileges()<br>• SQLTables() |
| **24**504 | The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row. | • SQLExecDirect()<br>• SQLExecute() |
| **25**000 **25**501 | Invalid transaction state. | • SQLDisconnect() |
| **34**000 | Invalid cursor name. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare()<br>• SQLSetCursorName() |
| **37***xxx* | Invalid SQL syntax. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLNativeSql()<br>• SQLPrepare() |
| **40**001 | Transaction rollback. | • SQLEndTran()<br>• SQLExecDirect()<br>• SQLExecute()<br>• SQLParamData()<br>• SQLPrepare() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **42***xxx* | Syntax error or access rule violation | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **42**000 | Invalid SQL syntax. | • SQLNativeSql() |
| **42**5*xx* | Syntax error or access rule violation | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **42**601 | PARMLIST syntax error. | • SQLProcedureColumns() |
| **42**818 | The operands of an operator or function are not compatible. | • SQLGetPosition() |
| **42**895 | The value of a host variable in the EXECUTE or OPEN statement cannot be used because of its data type | • SQLExecDirect()<br>• SQLExecute() |
| **42**S01[1] | Database object already exists. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **42**S02 | Database object does not exist. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **42**S11 | Index already exists. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **42**S12 | Index not found. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **42**S21 | Column already exists. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **42**S22 | Column not found. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLPrepare() |
| **44**000 | Integrity constraint violation. | • SQLExecDirect()<br>• SQLExecute() |
| **54**028 | The maximum number of concurrent LOB handles has been reached. | • SQLFetch() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **58**004 | Unexpected system failure. | • SQLBindCol()<br>• SQLBindFileToCol()<br>• SQLBindFileToParam()<br>• SQLBindParameter()<br>• SQLConnect()<br>• SQLDriverConnect()<br>• SQLDataSources()<br>• SQLDescribeCol()<br>• SQLDisconnect()<br>• SQLExecDirect()<br>• SQLExecute()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLFreeStmt()<br>• SQLGetCursorName()<br>• SQLGetData()<br>• SQLGetFunctions()<br>• SQLGetInfo()<br>• SQLGetLength()<br>• SQLGetPosition()<br>• SQLGetSubString()<br>• SQLMoreResults()<br>• SQLNumResultCols()<br>• SQLPrepare()<br>• SQLRowCount()<br>• SQLSetCursorName() |
| **HY**000[2] | General error. | • SQLAllocHandle()<br>• SQLCloseCursor()<br>• SQLColAttribute()<br>• SQLDescribeParam()<br>• SQLEndTran()<br>• SQLFreeHandle()<br>• SQLGetConnectAttr()<br>• SQLGetStmtAttr()<br>• SQLSetColAttributes()<br>• SQLSetConnection()<br>• SQLSetStmtAttr() |
| **HY**001 | Memory allocation failure. | All functions. |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **HY**002 | Invalid column number. | • SQLBindCol()<br>• SQLBindFileToCol()<br>• SQLColAttribute()<br>• SQLDescribeCol()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLGetData()<br>• SQLSetColAttributes() |
| **HY**003 | Program type out of range. | • SQLBindCol()<br>• SQLBindParameter()<br>• SQLGetData()<br>• SQLGetLength()<br>• SQLGetSubString() |
| **HY**004 | Invalid SQL data type. | • SQLBindParameter()<br>• SQLGetTypeInfo() |
| **HY**009 | Invalid use of a null pointer. | • SQLAllocHandle()<br>• SQLBindFileToCol()<br>• SQLBindFileToParam()<br>• SQLBindParameter()<br>• SQLColumnPrivileges()<br>• SQLExecDirect()<br>• SQLForeignKeys()<br>• SQLGetData()<br>• SQLGetFunctions()<br>• SQLGetInfo()<br>• SQLGetLength()<br>• SQLGetPosition()<br>• SQLNativeSql()<br>• SQLNumParams()<br>• SQLNumResultCols()<br>• SQLPrepare()<br>• SQLPutData()<br>• SQLSetCursorName()<br>• SQLSetConnectAttr()<br>• SQLSetEnvAttr()<br>• SQLSetStmtAttr() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **HY**010 | Function sequence error. | • SQLBindCol()<br>• SQLBindFileToCol()<br>• SQLBindFileToParam()<br>• SQLBindParameter()<br>• SQLCloseCursor()<br>• SQLColAttribute()<br>• SQLColumns()<br>• SQLDescribeCol()<br>• SQLDescribeParam()<br>• SQLDisconnect()<br>• SQLEndTran()<br>• SQLExecute()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLForeignKeys()<br>• SQLFreeHandle()<br>• SQLFreeStmt()<br>• SQLGetCursorName()<br>• SQLGetData()<br>• SQLGetFunctions()<br>• SQLGetStmtAttr()<br>• SQLGetTypeInfo()<br>• SQLMoreResults()<br>• SQLNumParams()<br>• SQLNumResultCols()<br>• SQLParamData()<br>• SQLParamOptions()<br>• SQLPrepare()<br>• SQLPrimaryKeys()<br>• SQLProcedureColumns()<br>• SQLProcedures()<br>• SQLPutData()<br>• SQLRowCount()<br>• SQLSetColAttributes()<br>• SQLSetConnectAttr()<br>• SQLSetCursorName()<br>• SQLSetStmtAttr()<br>• SQLSpecialColumns()<br>• SQLStatistics()<br>• SQLTablePrivileges()<br>• SQLTables() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **HY**011 | Operation invalid at this time. | • SQLSetConnectAttr()<br>• SQLSetEnvAttr()<br>• SQLSetStmtAttr() |
| **HY**012 | Invalid transaction code. | • SQLEndTran() |
| **HY**013 | Unexpected memory handling error. | • SQLAllocHandle()<br>• SQLBindCol()<br>• SQLBindFileToCol()<br>• SQLBindFileToParam()<br>• SQLBindParameter()<br>• SQLCancel()<br>• SQLCloseCursor()<br>• SQLConnect()<br>• SQLDataSources()<br>• SQLDescribeCol()<br>• SQLDisconnect()<br>• SQLExecDirect()<br>• SQLExecute()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLFreeHandle()<br>• SQLGetCursorName()<br>• SQLGetData()<br>• SQLGetFunctions()<br>• SQLGetLength()<br>• SQLGetPosition()<br>• SQLGetStmtAttr()<br>• SQLGetSubString()<br>• SQLMoreResults()<br>• SQLNumParams()<br>• SQLNumResultCols()<br>• SQLPrepare()<br>• SQLRowCount()<br>• SQLSetColAttributes()<br>• SQLSetCursorName() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **HY**014 | No more handles. | • SQLAllocHandle()<br>• SQLColumnPrivileges()<br>• SQLColumns()<br>• SQLExecDirect()<br>• SQLExecute()<br>• SQLForeignKeys()<br>• SQLPrepare()<br>• SQLPrimaryKeys()<br>• SQLProcedureColumns()<br>• SQLProcedures()<br>• SQLSpecialColumns()<br>• SQLStatistics()<br>• SQLTablePrivileges()<br>• SQLTables() |
| **HY**015 | No cursor name available. | • SQLGetCursorName() |
| **HY**019 | Numeric value out of range. | • SQLExecDirect()<br>• SQLExecute()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLGetData()<br>• SQLPutData() |
| **HY**024 | Invalid argument value. | • SQLConnect()<br>• SQLGetSubString()<br>• SQLSetConnectAttr()<br>• SQLSetEnvAttr()<br>• SQLSetStmtAttr() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **HY**090 | Invalid string or buffer length. | • SQLBindCol()<br>• SQLBindFileToCol()<br>• SQLBindFileToParam()<br>• SQLBindParameter()<br>• SQLColAttribute()<br>• SQLColumnPrivileges()<br>• SQLColumns()<br>• SQLConnect()<br>• SQLDataSources()<br>• SQLDescribeCol()<br>• SQLDriverConnect()<br>• SQLExecDirect()<br>• SQLParamData()<br>• SQLForeignKeys()<br>• SQLGetConnectAttr()<br>• SQLGetCursorName()<br>• SQLGetData()<br>• SQLGetInfo()<br>• SQLGetPosition()<br>• SQLGetStmtAttr()<br>• SQLGetSubString()<br>• SQLNativeSql()<br>• SQLPrepare()<br>• SQLPrimaryKeys()<br>• SQLProcedures()<br>• SQLProcedureColumns()<br>• SQLPutData()<br>• SQLSetColAttributes()<br>• SQLSetConnectAttr()<br>• SQLSetCursorName()<br>• SQLSetEnvAttr()<br>• SQLSetStmtAttr()<br>• SQLSpecialColumns()<br>• SQLStatistics()<br>• SQLTables()<br>• SQLTablePrivileges() |
| **HY**091 | Descriptor type out of range. | • SQLColAttribute() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **HY**092 | Option type out of range. | • SQLAllocHandle()<br>• SQLEndTran()<br>• SQLFreeStmt()<br>• SQLGetConnectAttr()<br>• SQLGetCursorName()<br>• SQLGetEnvAttr()<br>• SQLGetStmtAttr()<br>• SQLSetConnectAttr()<br>• SQLSetEnvAttr()<br>• SQLSetStmtAttr() |
| **HY**093 | Invalid parameter number. | • SQLBindFileToCol()<br>• SQLBindFileToParam()<br>• SQLBindParameter()<br>• SQLDescribeParam() |
| **HY**096 | Information type out of range. | • SQLGetInfo() |
| **HY**097 | Column type out of range. | • SQLSpecialColumns() |
| **HY**098 | Scope type out of range. | • SQLSpecialColumns() |
| **HY**099 | Nullable type out of range. | • SQLSpecialColumns() |
| **HY**100 | Uniqueness option type out of range. | • SQLStatistics() |
| **HY**101 | Accuracy option type out of range. | • SQLStatistics() |
| **HY**103 | Direction option out of range. | • SQLDataSources() |
| **HY**104 | Invalid precision value. | • SQLBindParameter()<br>• SQLSetColAttributes() |
| **HY**105 | Invalid parameter type. | • SQLBindParameter() |
| **HY**106 | Fetch type out of range. | • SQLExtendedFetch() |
| **HY**107 | Row value out of range. | • SQLParamOptions() |
| **HY**109 | Invalid cursor position. | • SQLGetStmtAttr() |
| **HY**110 | Invalid driver completion. | • SQLDriverConnect() |
| **HY**501 | Invalid data source name. | • SQLConnect() |
| **HY**506 | Error closing a file. | • SQLFreeHandle() |

*Table 265. SQLSTATE cross reference (continued)*

| SQLSTATE | Description | Functions |
|---|---|---|
| **HY**C00 | Driver not capable. | • SQLBindCol()<br>• SQLBindFileToCol()<br>• SQLBindFileToParam()<br>• SQLBindParameter()<br>• SQLColAttribute()<br>• SQLColumnPrivileges()<br>• SQLColumns()<br>• SQLDescribeCol()<br>• SQLDescribeParam()<br>• SQLExtendedFetch()<br>• SQLFetch()<br>• SQLForeignKeys()<br>• SQLGetConnectAttr()<br>• SQLGetData()<br>• SQLGetInfo()<br>• SQLGetLength()<br>• SQLGetPosition()<br>• SQLGetStmtAttr()<br>• SQLGetSubString()<br>• SQLPrimaryKeys()<br>• SQLProcedureColumns()<br>• SQLProcedures()<br>• SQLSetConnectAttr()<br>• SQLSetEnvAttr()<br>• SQLSetStmtAttr()<br>• SQLSpecialColumns()<br>• SQLStatistics()<br>• SQLTables()<br>• SQLTablePrivileges() |

**Notes:**

1. **42**S*xx* SQLSTATEs replace **S0**0*xx* SQLSTATEs.
2. **HY***xxx* SQLSTATEs replace **S1***xxx* SQLSTATEs.

**Related reference**

Deprecated ODBC functions and their replacements

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.

SQLSTATE mappings

Several SQLSTATEs differ when you call `SQLGetDiagRec()` or `SQLError()` under an ODBC 3.0 driver. All deprecated functions continue to return ODBC 2.0 SQLSTATEs regardless of which environment attributes are set.

**Related information**

ODBC functions
Db2 ODBC provides various SQL-related functions with unique purposes, diagnostics, and restrictions.

# Data conversion between the application and the database server

Data conversion is possible between C and SQL data types. You need to know the precision, scale, length, and display size of each data type. In addition, you will also need to know how to convert from one data type to the other.

Identifiers for date, time, and timestamp data types have also changed in ODBC 3.0.

**Related reference**

C and SQL data types
Db2 ODBC defines a set of SQL symbolic data types. Each SQL symbolic data type has a corresponding default C data type.

Data conversion
Db2 ODBC manages the transfer and any required conversion of data between the application and the database server. However, not all data conversions are supported.

Changes to datetime data types
The ODBC driver supports both ODBC 2.0 and ODBC 3.0 datetime values for input. For output, the ODBC driver determines the value to return based on the setting of the environment attribute.

## SQL data type attributes

SQL data type attributes include the precision, scale, length, and display size of this data type.

### Precision of SQL data types

The precision of a numeric column or parameter refers to the maximum number of digits that are used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter.

The following table defines the precision for each SQL data type.

*Table 266. Precision of SQL data types*

| fSqlType | Precision |
|---|---|
| SQL_CHAR<br>SQL_VARCHAR<br>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<br>SQL_NUMERIC | The defined maximum number of digits. For example, the precision of a column defined as NUMERIC(10,3) is 10. |
| SQL_DECFLOAT | 16 if the column is defined as DECFLOAT(16). 34 if the column is defined as DECFLOAT(34). |
| SQL_SMALLINT[2] | 5 |
| SQL_INTEGER[2] | 10 |
| SQL_BIGINT[2] | 19 |
| SQL_FLOAT[2] | 15 |

*Table 266. Precision of SQL data types (continued)*

| fSqlType | Precision |
|---|---|
| SQL_REAL[2] | 7 |
| SQL_ROWID | 40 |
| SQL_DOUBLE[2] | 15 |
| SQL_BINARY<br>SQL_VARBINARY<br>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). |
| SQL_TYPE_TIMESTAMP | The number of characters in the #yyyy-mm-dd hh:mm:ss[.fffffffffff]" or #yyyy-mm-dd.hh.mm.ss[.fffffffffff]" format that is 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 millionths 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 12 digits. |
| SQL_TYPE_TIME-<br>STAMP_WITH_TIMEZONE | |
| SQL_GRAPHIC SQL_VARGRAPHIC<br>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. |
| SQL_XML[2] | 0 |

**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.

The following table defines the scale for each SQL data type

*Table 267. Scale of SQL data types*

| fSqlType | Scale |
|---|---|
| SQL_CHAR<br>SQL_VARCHAR<br>SQL_LONGVARCHAR<br>SQL_CLOB | Not applicable. |

*Table 267. Scale of SQL data types (continued)*

| fSqlType | Scale |
|---|---|
| SQL_DECIMAL<br>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<br>SQL_INTEGER<br>SQL_BIGINT | 0 |
| SQL_DECFLOAT | Not applicable. |
| SQL_REAL<br>SQL_FLOAT<br>SQL_DOUBLE | Not applicable. |
| SQL_ROWID | Not applicable. |
| SQL_BINARY<br>SQL_VARBINARY<br>SQL_LONGVARBINARY<br>SQL_BLOB | Not applicable. |
| SQL_TYPE_DATE<br>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[fffffffffff]" 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 12 digits. You can retrieve the scale through the *pibScale* argument of `SQLDescribeCol()`. |
| SQL_GRAPHIC<br>SQL_VARGRAPHIC<br>SQL_LONGVARGRAPHIC<br>SQL_DBCLOB | Not applicable. |
| SQL_XML | 0 |

**Length of SQL data types**

The length of a column is the maximum number of bytes that are returned to the application when data is transferred to its default C data type.

For character data, the length does not include the null-termination character. Note that the length of a column can be different than the number of bytes that are required to store the data on the data source.

The following table defines the length for each SQL data type.

*Table 268. Length of SQL data types*

| fSqlType | Length |
|---|---|
| SQL_CHAR, SQL_VARCHAR<br>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. |

*Table 268. Length of SQL data types (continued)*

| fSqlType | Length |
|---|---|
| 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_BIGINT | 8 bytes |
| SQL_REAL | 4 bytes |
| SQL_ROWID | 40 bytes |
| SQL_FLOAT | 8 bytes |
| SQL_DOUBLE | 8 bytes |
| SQL_DECFLOAT | 23 bytes for DECFLOAT(16). 42 bytes for DECFLOAT(34). |
| 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 | 20 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. |
| SQL_XML | 0 bytes. However, stored XML documents are limited to a maximum size of 2 GB. |

**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.

The following table defines the display size for each SQL data type

*Table 269. 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_DECFLOAT | 23 bytes if the column is defined as DECFLOAT(16). 42 bytes if the column is defined as DECFLOAT(34). |
| 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). |

*Table 269. Display size of SQL data types (continued)*

| fSqlType | Display size |
|---|---|
| SQL_BIGINT | 20 bytes (a sign and 19 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). |
| 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 value is the number of characters in the #yyyy-mm-dd hh:mm:ss[.ffffffffffff] or #yyyy-mm-dd.hh.mm.ss[.ffffffffffff] format. For example, the display size of a column storing millionths of a second is 23 bytes (the number of characters in "yyyy-mm-dd hh:mm:ss.ffffff"). The maximum for fractional seconds is 12 digits. You can retrieve the display size of a timestamp column through the COLUMN_SIZE column that is returned by `SQLColumns()` or `SQLSpecialColumns()`. You can retrieve the display size of a SQL_TYPE_TIMESTAMP stored procedure parameter through the COLUMN_SIZE column that is returned by `SQLProcedureColumns()`. |
| 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. |
| SQL_XML | 0 |

## SQL to C data conversion

To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

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 07006 (restricted data type attribute violation).

If the *fCType* argument contains a value shown in the table but which specifies a conversion not supported by the driver, `SQLFetch()`, or `SQLGetData()` returns SQLSTATE HYC00 (driver not capable).

Though it is not shown in the tables, the *pcbValue* argument contains SQL_NULL_DATA when the SQL data value is null. For an explanation of the use of *pcbValue* when multiple calls are made to retrieve data, see `SQLGetData()`.

When SQL data is converted to character C data, the character count returned in *pcbValue* does not include the nul-termination character. If *rgbValue* is a null pointer, `SQLBindCol()` or `SQLGetData()` returns SQLSTATE HY009 (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.

**SQL to C conversion for character data**
The character SQL data types that you can convert to C data types are SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR, and SQL_CLOB.

The following table shows information about converting character SQL data to C data.

*Table 270. Converting character SQL data to C data*

| fCType | Test | rgbValue | pcbValue | SQLSTATE |
|---|---|---|---|---|
| SQL_C_CHAR<br>SQL_C_WCHAR | Data length < cbValueMax | Data | Data length (in bytes)"1" on page 548 | **00**000"2" on page 548 |
| | Data length >= cbValueMax | Truncated data | Data length (in bytes) | **01**004 |
| SQL_C_BINARY | Data length <= cbValueMax | Data | Data length (in bytes) | **00**000"2" on page 548 |
| | Data length > cbValueMax | Truncated data | Data length (in bytes) | **01**004 |
| SQL_C_DECIMAL64<br>SQL_C_DECIMAL128 | Data converted without truncation | Data | Data length (in bytes) | **00**000"1" on page 548 |
| | Data is not a number | Untouched | Data length (in bytes) | **22**005"3" on page 548 |

*Table 270. Converting character SQL data to C data (continued)*

| fCType | Test | rgbValue | pcbValue | SQLSTATE |
|---|---|---|---|---|
| SQL_C_SHORT<br>SQL_C_LONG<br>SQL_C_BIGINT<br>SQL_C_FLOAT<br>SQL_C_DOUBLE<br>SQL_C_TINYINT<br>SQL_C_BIT | Data converted without truncation"3" on page 548 | Data | Size (in bytes) of the C data type | **00**000"2" on page 548 |
| | Data converted with truncation, but without loss of significant digits"3" on page 548 | Data | Size (in bytes) of the C data type | **01**004 |
| | Conversion of data would result in loss of significant digits"3" on page 548 | Untouched | Size (in bytes) of the C data type | **22**003 |
| | Data is not a number"3" on page 548 | Untouched | Size (in bytes) of the C data type | **22**005 |
| SQL_C_TYPE_DATE | Data value is a valid date"3" on page 548 | Data | 6"4" on page 548 | **00**000"2" on page 548 |
| | Data value is not a valid date"3" on page 548 | Untouched | 6"4" on page 548 | **22**008 |
| SQL_C_TYPE_TIME | Data value is a valid time"3" on page 548 | Data | 6"4" on page 548 | **00**000"2" on page 548 |
| | Data value is not a valid time"3" on page 548 | Untouched | 6"4" on page 548 | **22**008 |
| SQL_C_TYPE_TIMESTAMP | Data value is a valid timestamp"3" on page 548 | Data | 16"4" on page 548 | **00**000"2" on page 548 |
| | Data value is not a valid timestamp"3" on page 548 | Untouched | 16"4" on page 548 | **22**008 |
| SQL_C_TYPE_TIME-STAMP_EXT | Data value is a valid timestamp"3" on page 548 | Data | 20"4" on page 548 | **00**000"2" on page 548 |
| | Data value is not a valid timestamp"3" on page 548 | Untouched | 20"4" on page 548 | **22**008 |

**Notes:**

1. For the SQL_C_WCHAR data type, the data length is the number of bytes of UCS-2 Unicode data.
2. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
3. 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.
4. This is the size of the corresponding C data type.

**Related reference**

SQL to C data conversion

To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

**SQL to C conversion for graphic data**

The graphic SQL data types that you can convert to C data types are SQL_GRAPHIC, SQL_VARGRAPHIC, SQL_LONGVARGRAPHIC, and SQL_DBCLOB.

The following table shows information about converting graphic SQL data to C data.

*Table 271. Conversion of 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<br>SQL_C_WCHAR | 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.

**Related reference**

SQL to C data conversion

To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

**SQL to C conversion for numeric data**

You can convert numeric SQL data types into C data types.

The numeric SQL data types are:

SQL_DECIMAL
SQL_DECFLOAT
SQL_NUMERIC
SQL_SMALLINT
SQL_INTEGER
SQL_BIGINT
SQL_REAL
SQL_FLOAT
SQL_DOUBLE

The following table shows information about converting numeric SQL data to C data.

*Table 272. Conversion of numeric SQL data to C data*

| fCType | Test | rgbValue | pcbValue | SQLSTATE |
|---|---|---|---|---|
| SQL_C_CHAR<br>SQL_C_WCHAR | 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<br>SQL_C_LONG<br>SQL_C_BIGINT<br>SQL_C_FLOAT<br>SQL_C_DOUBLE<br>SQL_C_TINYINT<br>SQL_C_BIT<br>SQL_C_DECIMAL64<br>SQL_C_DECIMAL128 | 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.

**Related reference**

SQL to C data conversion

To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

**SQL to C conversion for binary data**

You can convert binary SQL data types to C data types. The binary SQL data types are SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY, and SQL_BLOB.

The following table shows information about converting binary SQL data to C data.

*Table 273. Conversion of binary SQL data to C data*

| fCType | Test | rgbValue | pcbValue | SQLSTATE |
|---|---|---|---|---|
| SQL_C_CHAR<br>SQL_C_WCHAR | (Data length) < cbValueMax | Data | Data length (in bytes)"1" on page 550 | 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 |

**Note:**

1. For the SQL_C_WCHAR data type, the data length is the number of bytes of UCS-2 Unicode data.

**Related reference**

SQL to C data conversion
To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

**SQL to C conversion for date data**
You can convert the date SQL data type, SQL_TYPE_DATE, to a C data type.

The following table shows information about converting date SQL data to C data.

*Table 274. Conversion of 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_WCHAR | cbValueMax >= 22 | Data | 20 | **00**000[1] |
| | cbValueMax < 22 | Untouched | 20 | **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_TYPE_TIME-STAMP_EXT | None[2] | Data"5" on page 551 | 20[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.
5. The time fields of the TIMESTAMP_STRUCT_EXT structure are set to zero.

When the date SQL data type is converted to the character C data type, the resulting string is in the "yyyy-mm-dd" format.

**Related reference**

SQL to C data conversion
To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

**SQL to C conversion for time data**
You can convert the time SQL data type, SQL_TYPE_TIME, into a C data type.

The following table shows information about converting time SQL data to C data.

*Table 275. Conversion of 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_WCHAR | cbValueMax >= 18 | Data | 16 | **00**000[1] |
| | cbValueMax < 18 | Untouched | 16 | **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] |
| SQL_C_TYPE_ TIME-STAMP_EXT | None[2] | Data"5" on page 552 | 20[3] | **00**000[1] |

**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.
5. The time fields of the TIMESTAMP_STRUCT_EXT structure are 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.

**Related reference**

SQL to C data conversion

To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

**SQL to C conversion for timestamp data**

You can convert the timestamp SQL data type, SQL_TYPE_TIMESTAMP, to a C data type.

The following table shows information about converting timestamp SQL data to C data.

*Table 276. Conversion of 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 |

*Table 276. Conversion of timestamp SQL data to C data (continued)*

| fCType | Test | rgbValue | pcbValue | SQLSTATE |
|---|---|---|---|---|
| SQL_C_WCHAR | Display size < cbValueMax | Data | Data length (in bytes) | **00**000[1] |
| | 38 <= cbValueMax <= Display size | Truncated data[2] | Data length (in bytes) | **01**004 |
| | cbValueMax < 38 | 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 |
| QL_C_TYPE_TIME-STAMP_EXT | None [5] | Data | 20 | **00**000 |

**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).

**Related reference**

C to SQL data conversion
To convert C data types to SQL data types, you need to know the arguments: *fSqlType* and *pcbValue*. The SQLSTATE for each conversion outcome is returned.

**SQL to C conversion for ROWID data**

You can convert the ROWID SQL data type, SQL_ROWID, to a C data type.

The following table shows information about converting ROWID SQL data to C data.

*Table 277. Conversion of ROWID SQL data to C data*

| fCType | Test | rgbValue | pcbValue | SQLSTATE |
|---|---|---|---|---|
| SQL_C_CHAR<br>SQL_C_WCHAR | Data length < = cbValueMax | Data | Data length (in bytes) | **00**000 |
| | Data length > cbValueMax | Truncated data | Data length (in bytes) | **01**004 |

**Related reference**

SQL to C data conversion

To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

**SQL to C conversion for XML data**

You can convert the XML SQL data type, SQL_XML, to a C data type.

The following table shows information about converting XML SQL data to C data.

*Table 278. Conversion of XML SQL data to C data*

| fCType | Test | rgbValue | pcbValue | SQLSTATE |
|---|---|---|---|---|
| SQL_C_CHAR | Data length < = cbValueMax | Data | Data length (in bytes) | **00**000[“1” on page 554] |
| | 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” on page 554] |
| | Data length > cbValueMax | Truncated data | Data length (in bytes) | **01**004 |
| SQL_C_BINARYXML | Data length < = cbValueMax | Data | Data length (in bytes) | **00**000[“1” on page 554] |
| | Data length > cbValueMax | Truncated data | Data length (in bytes) | **01**004 |
| SQL_C_DBCHAR | Number of double-byte characters * 2 < cbValueMax | Data | Data length (in bytes) | **00**000[“1” on page 554] |
| | Number of double-byte characters * 2 >= cbValueMax | Truncated data, to the nearest even byte that is less than cbValueMax | Data length (in bytes) | **01**004 |
| SQL_C_WCHAR | Number of double-byte characters * 2 < cbValueMax | Data | Data length (in bytes) | **00**000[“1” on page 554] |
| | Number of double-byte characters * 2 >= cbValueMax | Truncated data, to the nearest even byte that is less than cbValueMax | Data length (in bytes) | **01**004 |

**Note:**

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

**Related reference**

SQL to C data conversion

To convert SQL data types to C data types, you need to know the arguments: *fCType* , *cbValueMax*, *rgbValue*, and *pcbValue*. The SQLSTATE for each conversion outcome isreturned.

**SQL to C data conversion examples**

The SQL data types you can convert to C data types are character, graphic, numeric, binary, date, time, timestamp, ROWID, and XML data.

The following table shows example SQL to C data conversions and the SQLSTATE values that are associated with these conversions.

*Table 279. 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.

## C to SQL data conversion

To convert C data types to SQL data types, you need to know the arguments: *fSqlType* and *pcbValue*. The SQLSTATE for each conversion outcome is returned.

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 00000 (success).

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

If the *fSqlType* argument in `SQLBindParameter()` contains a value not shown in the table for a given C data type, SQLSTATE 07006 is returned (Restricted data type attribute violation).

If the *fSqlType* argument contains a value shown in the table but which specifies a conversion not supported by the driver, `SQLBindParameter()` returns SQLSTATE HYC00 (Driver not capable).

If the *rgbValue* and *pcbValue* arguments specified in `SQLBindParameter()` are both null pointers, that function returns SQLSTATE HY009 (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.

### C to SQL conversion for character data

You can convert the C data types, SQL_C_CHAR and SQL_C_WCHAR, to SQL data types.

The data length for the SQL_C_WCHAR data type is the number of bytes of UCS-2 Unicode data.

The following table shows information about converting character C data to SQL data.

*Table 280. Conversion of character C data to SQL data*

| fSqlType | Test | SQLSTATE |
|---|---|---|
| SQL_CHAR<br>SQL_VARCHAR<br>SQL_LONGVARCHAR<br>SQL_CLOB | Data length <= Column length | **00**000[1] |
| | Data length > Column length | **01**004 |

*Table 280. Conversion of character C data to SQL data (continued)*

| fSqlType | Test | SQLSTATE |
|---|---|---|
| SQL_DECIMAL<br>SQL_NUMERIC<br>SQL_SMALLINT<br>SQL_INTEGER<br>SQL_BIGINT<br>SQL_REAL<br>SQL_FLOAT<br>SQL_DOUBLE<br>SQL_DECFLOAT | 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<br>SQL_VARBINARY<br>SQL_LONGVARBINARY<br>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[1] |
| SQL_TYPE_TIMESTAMP | Data value is a valid timestamp | **00**000[1] |
| | Data value is not a valid timestamp. | **22**008 |
| | Data value is a valid timestamp; fractional seconds portion is nonzero | **01**004 |
| SQL_GRAPHIC<br>SQL_VARGRAPHIC<br>SQL_LONGVARGRAPHIC<br>SQL_DBCLOB | Data length / 2 <= Column length | **00**000[1] |
| | Data length / 2 < Column length | **01**004 |
| SQL_XML | None | **00**000[1] |

**Note:**

1. SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

**Related reference**

C to SQL data conversion
To convert C data types to SQL data types, you need to know the arguments: *fSqlType* and *pcbValue*. The SQLSTATE for each conversion outcome is returned.

**C to SQL conversion for numeric data**
You can convert numeric C data types to SQL data types.

The numeric C data types are:

SQL_C_SHORT
SQL_C_LONG
SQL_C_BIGINT
SQL_C_FLOAT
SQL_C_DOUBLE
SQL_C_TINYINT
SQL_C_BIT
SQL_C_DECIMAL64
SQL_C_DECIMAL128

The following table shows information about converting numeric C data to SQL data.

*Table 281. Conversion of numeric C data to SQL data*

| fSqlType | Test | SQLSTATE |
|---|---|---|
| SQL_DECIMAL<br>SQL_NUMERIC<br>SQL_SMALLINT<br>SQL_INTEGER<br>SQL_BIGINT<br>SQL_REAL<br>SQL_FLOAT<br>SQL_DOUBLE<br>SQL_DECFLOAT | 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.

**Related reference**

C to SQL data conversion
To convert C data types to SQL data types, you need to know the arguments: *fSqlType* and *pcbValue*. The SQLSTATE for each conversion outcome is returned.

**C to SQL conversion for binary data**

You can convert the binary C data types, SQL_C_BINARY and SQL_C_BINARYXML, to SQL data types.

The following table shows information about converting C data of type SQL_C_BINARY to SQL data.

*Table 282. Conversion of SQL_C_BINARY 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 |
| SQL_XML | None | **00**000[1] |

*Table 282. Conversion of SQL_C_BINARY data to SQL data (continued)*

| fSqlType | Test | SQLSTATE |
|---|---|---|

**Note:**

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

The following table shows information about converting C data of type SQL_C_BINARYXML to SQL data.

*Table 283. Conversion of SQL_C_BINARYXML data to SQL data*

| fSqlType | Test | SQLSTATE |
|---|---|---|
| SQL_XML | None | **00**000[1] |

**Note:**

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

**Related reference**

C to SQL data conversion
To convert C data types to SQL data types, you need to know the arguments: *fSqlType* and *pcbValue*. The SQLSTATE for each conversion outcome is returned.

**C to SQL conversion for double-byte character data**

You can convert the double-byte C data type, SQL_C_DBCHAR, to an SQL data type.

The following table shows information about converting double-byte character C data to SQL data.

*Table 284. Conversion of 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 |
| SQL_XML | None | **00**000[1] |

**Note:**

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

**Related reference**

C to SQL data conversion
To convert C data types to SQL data types, you need to know the arguments: *fSqlType* and *pcbValue*. The SQLSTATE for each conversion outcome is returned.

**C to SQL conversion for date data**

You can convert the date C data type, SQL_C_TYPE_DATE, to an SQL data type.

The following table shows information about converting date C data to SQL data.

*Table 285. 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.

**Related reference**

C to SQL data conversion
To convert C data types to SQL data types, you need to know the arguments: *fSqlType* and *pcbValue*. The SQLSTATE for each conversion outcome is returned.

**C to SQL conversion for time data**

You can convert the time C data type, SQL_C_TYPE_TIME, to an SQL data type.

The following table shows information about converting time C data to SQL data.

*Table 286. Conversion of 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.

**Related reference**

C to SQL data conversion
To convert C data types to SQL data types, you need to know the arguments: *fSqlType* and *pcbValue*. The SQLSTATE for each conversion outcome is returned.

**C to SQL conversion for timestamp data**

You can convert the timestamp C data type, SQL_C_TYPE_TIMESTAMP, to an SQL data type.

The following table shows information about converting timestamp C data to SQL data.

*Table 287. Conversion of timestamp C data to SQL data*

| fSqlType | Test | SQLSTATE |
|---|---|---|
| SQL_CHAR<br>SQL_VARCHAR | Column length >= Display size | **00**000"1" on page 561 |
| | 19 <= Column length < Display size | **01**004 |
| | Column length < 19 | **22**003 |
| | Fractional seconds field length > 12 (display size > 32) | **22**008 |
| SQL_TYPE_DATE | Data value is a valid date, and time fields are 0 | **00**000"1" on page 561 |
| | Data value is a valid date, and time fields are not 0"2" on page 561 | **01**004 |
| | Data value is not a valid date | **22**008 |
| SQL_TYPE_TIME | Data value is a valid time. Fractional seconds fields are zero. | **00**000"1" on page 561 |
| | Data value is a valid time. Fractional seconds fields are not zero. | **01**004 |
| | Data value is not a valid time. | **22**008 |
| SQL_TYPE_TIMESTAMP | Data value is a valid timestamp | **00**000"1" on page 561 |
| | Data value is not a valid timestamp | **22**008 |
| | Precision specified by TIMESTAMP(*p*) < fractional seconds field length <= 12"4" on page 561 | **00**000"1" on page 561 |

**Notes:**

1. SQLSTATE **00**000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.
2. The time portion of the timestamp is deleted.
3. The date portion of the timestamp is deleted.
4. Fractional seconds of the timestamp are truncated.
5. The time zone component of TIMESTAMP is set based on either CLIENTTIMEZONE, SESSIONTIMEZONE, or the current system time zone of the machine the application is running.

**Related reference**

C to SQL data conversion
To convert C data types to SQL data types, you need to know the arguments: *fSqlType* and *pcbValue*. The SQLSTATE for each conversion outcome is returned.

**C to SQL data conversion examples**

The C data types that you can convert to SQL data types are character, numeric, binary, double-byte, date, time, and timestamp data.

The following table shows example C to SQL data conversions and the SQLSTATE associated with these conversions.

*Table 288. 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 |

*Table 288. C to SQL data conversion examples (continued)*

| C data type | C data Value | SQL data type | Column length | SQL data value | SQLSTATE |
|---|---|---|---|---|---|
| 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.

# Deprecated ODBC functions

Db2 ODBC supports the ODBC 3.0 standard and all deprecated functions. Use the function replacements where applicable to optimize performance.

This information explains the Db2 for z/OS support of the ODBC 3.0 standard.

## Deprecated ODBC functions and their replacements

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.

The following table lists the ODBC 2.0 deprecated functions and the ODBC 3.0 replacement functions.

*Table 289. 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() |

*Table 289. ODBC 2.0 deprecated functions (continued)*

| ODBC 2.0 deprecated function | Purpose | ODBC 3.0 replacement function |
|---|---|---|
| SQLGetStmtOption() | Returns a value of a statement attribute. | SQLGetStmtAttr() |
| SQLParamOptions() | Sets multiple values at one time for each bound parameter. | SQLSetStmtAttr() |
| 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() InfoType argument values

Values of the *InfoType* arguments for SQLGetInfo() arguments are renamed in ODBC 3.0.

The following table shows the renamed SQLGetInfo() InfoTypes in ODBC 3.0.

*Table 290. 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 |
| 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 |

**Related reference**

SQLGetInfo() - Get general information
SQLGetInfo() returns general information about the database management systems to which the application is currently connected. For example, SQLGetInfo() indicates which data conversions are supported.

## Changes to SQLSetConnectAttr() attributes

For SQLSetConnectAttr() attributes, the ODBC driver supports both ODBC 2.0 and ODBC 3.0 values.

The following table correlates ODBC 2.0 and ODBC 3.0 values.

*Table 291. 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

For SQLSetEnvAttr() attributes, the ODBC driver supports both ODBC 2.0 and ODBC 3.0 values.

The following table lists the SQLSetEnvAttr() attribute values renamed in ODBC 3.0. The ODBC 3.0 attributes support all of the existing ODBC 2.0 attributes.

*Table 292. 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

For SQLSetStmtAttr() attributes, the ODBC driver supports both ODBC 2.0 and ODBC 3.0 values.

The following table lists the SQLSetStmtAttr() attribute values renamed in ODBC 3.0. The ODBC 3.0 attributes support all of the existing ODBC 2.0 attributes.

*Table 293. 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 |
| 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 |

*Table 293. SQLSetStmtAttr() attribute value mapping (continued)*

| ODBC 2.0 attribute | ODBC 3.0 attribute |
|---|---|
| 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 that is 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 ODBC 3.0driver 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 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 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 ODBC 2.0driver 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 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 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 ODBC 3.0 driver.

## SQLSTATE mappings

Several SQLSTATEs differ when you call SQLGetDiagRec() or SQLError() under an ODBC 3.0 driver. All deprecated functions continue to return ODBC 2.0 SQLSTATEs regardless of which environment attributes are set.

The following list shows the affected SQLSTATEs:

- **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.

The following table lists ODBC 2.0 to ODBC 3.0 SQLSTATE mappings.

*Table 294. ODBC 2.0 to ODBC 3.0 SQLSTATE mappings*

| ODBC 2.0 SQLSTATE | ODBC 3.0 SQLSTATE |
|---|---|
| **22**003 | **HY**019 |
| **22**007 | **22**008 |
| **22**005 | **22**018 |
| **37**000 | **42**000 |
| **S0**001 | **42**S01 |
| **S0**002 | **42**S02 |
| **S0**011 | **42**S11 |
| **S0**012 | **42**S12 |
| **S0**021 | **42**S21 |
| **S0**022 | **42**S22 |
| **S0**023 | **42**S23 |
| **S1**000 | **HY**000 |
| **S1**001 | **HY**001 |
| **S1**002 | **HY**002 |
| **S1**003 | **HY**003 |
| **S1**004 | **HY**004 |
| **S1**009 | **HY**009 or **HY**024<br><br>**S1**009 is mapped to **HY**009 for invalid use of null pointers;<br>**S1**009 is mapped to **HY**024 for invalid attribute values. |
| **S1**010 | **HY**010 |
| **S1**011 | **HY**011 |
| **S1**012 | **HY**012 |
| **S1**013 | **HY**013 |
| **S1**014 | **HY**014 |
| **S1**015 | **HY**015 |
| **S1**019 | **HY**019 |
| **S1**090 | **HY**090 |
| **S1**091 | **HY**091 |
| **S1**092 | **HY**092 |
| **S1**093 | **HY**093 |
| **S1**096 | **HY**096 |
| **S1**097 | **HY**097 |
| **S1**098 | **HY**098 |
| **S1**099 | **HY**099 |

*Table 294. ODBC 2.0 to ODBC 3.0 SQLSTATE mappings (continued)*

| ODBC 2.0 SQLSTATE | ODBC 3.0 SQLSTATE |
|---|---|
| **S1**100 | **HY**100 |
| **S1**101 | **HY**101 |
| **S1**103 | **HY**103 |
| **S1**104 | **HY**104 |
| **S1**105 | **HY**105 |
| **S1**106 | **HY**106 |
| **S1**107 | **HY**107 |
| **S1**110 | **HY**110 |
| **S1**501 | **HY**501 |
| **S1**506 | **HY**506 |
| **S1**C00 | **HY**C00 |

## Changes to datetime data types

The ODBC driver supports both ODBC 2.0 and ODBC 3.0 datetime values for input. For output, the ODBC driver determines the value to return based on the setting of the environment attribute.

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 295 on page 567 for SQL type mappings and Table 296 on page 567 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.

The following figures show the corresponding ODBC 2.0 to ODBC 3.0 mappings for SQL type and C type identifiers respectively.

*Table 295. 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_TIMESTAMP(93) |

*Table 296. 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) |

*Table 296. Datetime data type mappings: C type identifiers (continued)*

| ODBC 2.0 | ODBC 3.0 |
|---|---|
| SQL_C_TIMESTAMP_EXT | SQL_C_TYPE_TIMESTAMP_EXT |

The datetime data type changes affect the following functions:

- SQLBindCol()
- SQLBindParameter()
- SQLColAttribute()
- SQLColumns()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLGetData()
- SQLGetTypeInfo()
- SQLProcedureColumns()
- SQLStatistics()
- SQLSpecialColumns()

# Example Db2 ODBC code

You can view example Db2 ODBC code for a sample verification program DSN8O3VP and for a client application APD29 that calls a stored procedure.

The following sample applications are provided:

- DSN8O3VP, which is also available online in the DSN810.SDSNSAMP data set can be used to verify that your Db2 ODBC 3.0 installation is correct.
- APD29, a client application that calls a Db2 ODBC stored procedure (SPD29), 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.

**Related concepts**

DSN8O3VP sample application
The DSN8O3VP sample program validates the installation of Db2 ODBC.

Client application calling a Db2 ODBC stored procedure
The client application, APD29, calls the stored procedure, SPD29, and processes query result sets. A query cursor opens in a stored procedure and returns to the client for fetching.

**Related tasks**

Preparing and executing an ODBC application
You must compile, prelink, and link-edit an ODBC application before you can run it. Db2 ODBC provides sample programs to help you with program preparation.

## DSN8O3VP sample application

The DSN8O3VP sample program validates the installation of Db2 ODBC.

The following code contains the sample program DSN810.SDSNSAMP(DSN8O3VP).

```
/*********************************************************************/
/*  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;}
```

```
   /*****************************************************************/
   /* 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");
```

```
 /******** 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*/
```

```
/*****************************************************************/
/* 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 */
```

```
/*********************************************************************/
/* 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 */
```

```
/*********************************************************************/
/* 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                                                      */
/*********************************************************************/
```

## Client application calling a Db2 ODBC stored procedure

The client application, APD29, calls the stored procedure, SPD29, and processes query result sets. A query cursor opens in a stored procedure and returns to the 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=
    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=
    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=
      strcpy((char *)sqlstmt,"insert into table2a values(");
      sprintf((char *)sqlstmt+strlen((char *)sqlstmt),"
      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=
      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
  WLM ENVIRONMENT WLMENV1;
```

**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];
```

```
   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;
```

```
   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 =
   printf("SPD29 received =
   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 result =
    printf("SPD29 expected =
    printf("SPD29 received spname=
    printf("SPD29 received argv0 =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=
 /*************************************************************/
 printf("\nSPD29 SQLAllocConnect                              ");
 hdbc=0;
```

```
SQLAllocConnect(henv, &hdbc);
if( rc != SQL_SUCCESS ) goto dberror;
printf("\nSPD29-hdbc=
```

```
/*****************************************************************/
/* 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.SYSROUTINES,   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=
printf("\nSPD29 uid=
printf("\nSPD29 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=

    printf("\nSPD29 SQLPrepare                                  \n");
    strcpy((char *)sqlstmt,
    "insert into TABLE2A(int4) values(?)");
    printf("\nSPD29 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=
       goto dberror;
      }
```

```
    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=
```

```
    hstmt=0;
    rc=SQLAllocStmt(hdbc, &hstmt);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPDIN-hstmt=

    jx=201;
    printf("\nAPDIN SQLExecDirect  stmt=
    strcpy((char *)sqlstmt,"insert into table2a values(");
    sprintf((char *)sqlstmt+strlen((char *)sqlstmt),"
    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=
    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=

    printf("\nSPD29 SQLPrepare                                   \n");
    strcpy((char *)sqlstmt,
     "SELECT INT4 FROM TABLE2A");
    printf("\nSPD29 sqlstmt=
    rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nSPD29 SQLExecute                                  \n");
    rc=SQLExecute(hstmt);
    if( rc != SQL_SUCCESS ) goto dberror;
```

```
  /* 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=

    printf("\nSPD29 SQLPrepare                                   \n");
    strcpy((char *)sqlstmt2,
     "SELECT CHR10 FROM TABLE2A");
    printf("\nSPD29 sqlstmt2=
    rc=SQLPrepare(hstmt2,sqlstmt2,SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nSPD29 SQLExecute                                  \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=
     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=
 check_error(henv,hdbc,hstmt,rc);
 printf("\nSPD29 SQLFreeEnv         number=     7\n");
 rc = SQLFreeEnv(henv);
```

```
 printf("\nSPD29 rc=
 rc=12;
 rc=12;
 SPCODE=12;
 goto pgmend;

 pgmend:

 printf("\nSPD29  TERMINATION   ");
 if (rc!=0)
  {
  printf("\nSPD29 WAS NOT SUCCESSFUL");
  printf("\nSPD29 SPCODE  =
  printf("\nSPD29 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=
        break;
    } /*end switch*/
```

```
    printf("\nSPD29 SQLError  ");
    while ((rc=SQLError(henv, hdbc, hstmt, cli_sqlstate, &cli_sqlcode,
      buffer,SQL_MAX_MESSAGE_LENGTH + 1, &length)) == SQL_SUCCESS) {
        printf("         SQLSTATE:
        printf("Native Error Code:
        printf("
    };
    if (rc!=SQL_NO_DATA_FOUND)
      printf("SQLError api call failed 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=

    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 ....
    printf("\nSQLCABC ....
    printf("\nSQLCODE ....
    printf("\nSQLERRML ...
    printf("\nSQLERRMC ...
    printf("\nSQLERRP  ...
    for (i = 0; i < 6; i++)
      printf("\nSQLERRD
    for (i = 0; i < 10; i++)
      printf("\nSQLWARNi]);
    printf("\nSQLWARNA ... 10]);
    printf("\nSQLSTATE ...

    return(0);
  }                                                 /* End of prtsqlca */
  /*********************************************************************/
  /*END OF SPD29 ******************************************************/
```

**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;
```

```
   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;
```

```
   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=
    henv=0;
    rc = SQLAllocEnv(&henv);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPD29-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=
 /***************************************************************/
    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=
    printf("\nAPD29 uid=
    printf("\nAPD29 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=
hstmt=0;
rc=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
printf("\nAPD29-hstmt=

SPCODE=0;
printf("\nAPD29 call sp SPCODE =
printf("\nAPD29 SQLPrepare  stmt=
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLBindParameter  stmt=
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=
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
if( SPCODE != 0 )
{
  printf("\nAPD29 SPCODE not zero, spcode=
  goto dberror;
}

printf("\nAPD29 SQLTransact stmt=
rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29  SQLFreeStmt stmt=
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=
hstmt=0;
rc=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
printf("\nAPD29-hstmt=
```

```
SPCODE=1;
printf("\nAPD29 call sp SPCODE =
printf("\nAPD29 SQLPrepare  stmt=
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLBindParameter  stmt=
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=
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
if( SPCODE != 0 )
{
  printf("\nAPD29 SPCODE not zero, spcode=
  goto dberror;
```

```
}

printf("\nAPD29 SQLTransact stmt=
rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29  SQLFreeStmt stmt=
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=

SPCODE=2;
printf("\nAPD29 call sp SPCODE =
printf("\nAPD29 SQLPrepare           number=   19\n");
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;
```

```
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=
   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 =
 if ( (H1INT4<=0) ||  (H1INT4>=202)
      || (LNH1INT4!=4 && LNH1INT4!=-1)  )
  { /* data error */
    printf("\nAPD29 H1INT4=
    printf("\nAPD29 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;
 }
```

```
 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=
   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 =
   result=strcmp((char *)H1CHR10,"CHAR      ");
   if ( (result!=0)
       || (LNH1INT4!=4 && LNH1INT4!=-1)  )
   {
     printf("\nAPD29 H1CHR10=
     printf("\nAPD29 result=
     printf("\nAPD29 LNH1CHR10=
     printf("\nAPD29 strlen(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=
    rc=SQLDisconnect(hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;
 /*****************************************************************/
    printf("\nSQLFreeConnect  stmt=
    rc=SQLFreeConnect(hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;
  /*****************************************************************/
  /* End SQL statements ********************************************/
```

```
} /* 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=
 /*****************************************************************/
```

```
    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=
    printf("\nAPD29 uid=
    printf("\nAPD29 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=

    printf("\nAPD01 SQLExecDirect                                 \n");
    strcpy((char *)sqlstmt,
    "SELECT INT4 FROM TABLE2A WHERE INT4=200");
    printf("\nAPD01 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=
     printf("\nAPD01 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_CLOSE);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nAPD01 SQLExecDirect                                 \n");
    strcpy((char *)sqlstmt,
    "SELECT INT4 FROM TABLE2A WHERE INT4=201");
    printf("\nAPD01 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=
     printf("\nAPD01 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=
    rc=SQLDisconnect(hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;
 /*****************************************************************/
    printf("\nSQLFreeConnect  stmt=
    rc=SQLFreeConnect(hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;

} /* end for each site perform these stmts */
 /*****************************************************************/
    printf("\nSQLFreeEnv  stmt=
    rc=SQLFreeEnv(henv);
    if( rc != SQL_SUCCESS ) goto dberror;
 /*****************************************************************/
 goto pgmend;

 dberror:
 printf("\nAPD29 entry dberror label");
 printf("\nAPD29 rc=
 check_error(henv,hdbc,hstmt,rc);
 printf("\nAPDXX SQLFreeEnv         number=     6\n");
 rc=SQLFreeEnv(henv);
 printf("\nAPDXX FREEENV rc =
 rc=12;
 printf("\nAPDXX DBERROR set rc =
 goto pgmend;
```

```
 pgmend:
     printf("\nAPD29  TERMINATION   ");
     if (rc!=0)
      {
      printf("\nAPD29 WAS NOT SUCCESSFUL");
      printf("\nAPD29 SPCODE  =
      printf("\nAPD29 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=
        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:
        printf("Native Error Code:
        printf("
    };
    if (rc!=SQL_NO_DATA_FOUND)
      printf("SQLError api call failed rc=
```

```
    printf("\nAPD29 SQLGetSQLCA  ");
    rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    if( rc == SQL_SUCCESS )
      prt_sqlca();
    else
      printf("\n  APD29-check_error SQLGetSQLCA failed 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 ....
    printf("\nSQLCABC ....
    printf("\nSQLCODE ....
    printf("\nSQLERRML ...
    printf("\nSQLERRMC ...
    printf("\nSQLERRP  ...
    for (i = 0; i < 6; i++)
      printf("\nSQLERRD
    for (i = 0; i < 10; i++)
      printf("\nSQLWARNi]);
    printf("\nSQLWARNA ... 10]);
    printf("\nSQLSTATE ...

    return(0);
  }                                              /* End of prtsqlca */
  /*END OF APD29*******************************************************/
```

# Information resources for Db2 11 for z/OS and related products

Information about Db2 11 for z/OS and products that you might use in conjunction with Db2 11 is available online in .

You can find the complete set of product documentation for Db2 11 for z/OS in IBM Knowledge Center.

You can also download other PDF format manuals for Db2 11 for z/OS from in PDF format manuals for Db2 11 for z/OS (Db2 Home).

© Copyright IBM Corp. 1997, 2020

# Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing IBM Corporation*
*North Castle Drive, MD-NC119*
*Armonk, NY 10504-1785 US*

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing Legal and Intellectual Property Law IBM Japan Ltd.*
*19-21, Nihonbashi-Hakozakicho, Chuo-ku*
*Tokyo 103-8510, Japan*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*IBM Director of Licensing IBM Corporation*
*North Castle Drive, MD-NC119*
*Armonk, NY 10504-1785 US*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

© Copyright IBM Corp. 1997, 2020

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as shown below:

© (*your company name*) (*year*).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. (*enter the year or years*).

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

## Programming interface information

This information is intended to help you write applications that use ODBC to access Db2 11 for z/OS servers. This information primarily documents General-use Programming Interface and Associated Guidance Information provided by Db2 11 for z/OS.

#### General-use Programming Interface and Associated Guidance Information

General-use Programming Interfaces allow the customer to write programs that obtain the services of Db2 11 for z/OS.

## Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered marks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at: http://www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

## Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions:

**Applicability:** These terms and conditions are in addition to any terms of use for the IBM website.

**Personal use:** You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

**Commercial use:** You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

**Rights:** Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

## Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

# Glossary

The glossary is available in IBM Knowledge Center.

See the Glossary topic for definitions of Db2 for z/OS terms.

© Copyright IBM Corp. 1997, 2020

# Index

## Special Characters


_ 421
% 421


## Numerics


64-bit applications
  compiling 48
  link-editing 52
64-bit ODBC applications
  example 76
  migrating to 76
64-bit ODBC driver 39


## A


abends 519
abnormal termination 519
accessibility
  keyboard x
  shortcut keys x
add rows
  SQLBulkOperations() 121
allocate handles
  initialization and termination 10
  SQLAllocHandle() 87
  transaction processing 14
application
  bind variables 480
  compile 46, 47
  compiling XPLINK 48
  DLL 45
  encoding schemes 479, 485
  execute 53
  execution steps 46
  link-edit 50
  multithreaded 471, 473
  prelink 50
  preparation 44
  requirements 45
  sample, DSNTEJ8 46
  tasks 9
  trace 507
  trace, obtaining 517, 518
application trace 507
application trace, obtaining 518
application variables, binding 17
array input 424
array output 428
arrays
  sending and retrieving data 505
ASCII 479
ASCII scalar function 525
ATRBEG service 418
ATREND service 418
ATRSENV service 418
attributes
  connection 411, 413
  environment 411
  querying and setting 411
  retrieving
    SQLColAttribute() 129
    SQLDescribeCol() 157
    SQLGetConnectAttr() 220
    SQLGetEnvAttr() 242
    SQLGetStmtAttr() 287
  setting
    SQLSetColAttributes() 346
    SQLSetConnectAttr() 350
    SQLSetEnvAttr() 367
    SQLSetStmtAttr() 378
  statement 411, 413
  unit of work semantics 416
authentication 149
autocommit mode 21


## B


batch processing 315
BIGINT
  conversion to C 549
  length 544
  precision 542
  scale 543
binary
  conversion to C 550
BINARY
  length 544
  precision 542
  scale 543
binary data conversion 550
bind functions
  SQLBindCol() 92
  SQLBindFileToCol() 99
  SQLBindFileToParam() 103
  SQLBindParameter() 106
binding
  application variables
    columns 18
    parameter markers 17
  DBRMs 40
  decimal floating point data 454
  options for ODBC applications 41
  plan 42
  return codes 42
  sample, DSNTIJCL 39, 42
  stored procedures 40
  variable-length timestamp data 455
bulk inserts
  SQLBulkOperations() 447, 453

## C


caching
  dynamic SQL statement 503
call level interface 2
cancel function
  SQLCancel() 125
case sensitivity 35
catalog
  functions
    example 422
    input argument 421
    limiting use of 502
    overview 420
    SQLColumnPrivileges() 138
    SQLColumns() 142
    SQLPrimaryKeys() 323
    SQLProcedureColumns() 327
    SQLProcedures() 336
    SQLSpecialColumns() 390
    SQLStatistics() 396
    SQLTablePrivileges() 401
    SQLTables() 405
  querying 420
CHAR
  conversion to C 547
  display size 545
  length 544
  precision 542
  scale 543
character strings 34, 35
CLISCHEMA keyword
  using 423
close cursor 217
column attributes
  retrieving 129
  setting 346
column functions
  SQLColAttribute() 129
  SQLColumnPrivileges() 138
  SQLColumns() 142
  SQLDescribeCol() 157
  SQLForeignKeys() 206
  SQLNumResultCols() 310
  SQLPrimaryKeys() 323
  SQLSetColAttributes() 346
  SQLSpecialColumns() 390
column-wise binding 428, 429
commit 21, 172
common server 2
compile, application 46, 47
compile, non-XPLINK applications 47
compiling
  64-bit applications 48
CONCAT scalar function 525
configuring
  application 44
configuring Db2 ODBC 37
connect functions
  SQLConnect() 149
  SQLDriverConnect() 167
connecting to a data source
  multiple data sources 13
connection string 411
connections
  active 362
  attributes
    changing 411
    setting and retrieving 413
    SQLGetConnectAttr() 220
    SQLSetConnectAttr() 350
  handles
    allocating 10
    description 4
    freeing 10
    SQLAllocHandle() 87
    SQLFreeHandle() 215
  type 12
connectivity
  ODBC model 11
  requirements 38
contexts
  multiple 473
  private external 477
coordinated distributed transactions 414
coordinated transactions, establishing 414
core level functions 1
CURDATE scalar function 526
CURRENTDATA, bind option for ODBC applications 41
CURRENTSERVER, plan bind option 42
cursors
  definition 20
  functions
    SQLCloseCursor() 127
    SQLFreeStmt() 217
    SQLGetCursorName() 223
    SQLSetCursorName() 364
  hold behavior 502
  use in Db2 ODBC 4
CURTIME scalar function 526


## D


data conversion
  C data types 25
  C to SQL data types 556
  data types 25
  default data types 25
  description 31
  display size of SQL data types 545
  length of SQL data types 544
  overview 542
  precision of SQL data types 542
  scale of SQL data types 543
  SQL data types 25
  SQL to C data types 546
  supported data types 31
data conversion, examples 555, 561
data retrieval, guidelines 502
data source information, querying 35
data sources
  general information 249
  get functions
    SQLGetInfo() 249
  SQLDataSources() 154
  supported data types 295
data structure, SQLCA 283
data types

data types *(continued)*
C 25, 30
generic 30
ODBC 30
SQL 25
SQLGetTypeInfo() 295
data-at-execute 451
DATABASE scalar function 526
DATE
conversion to C 551
display size 545
length 544
precision 542
scale 543
date and time function 526
DATE_STRUCT 25
datetime data types, changes 567
DAYOFMONTH scalar function 526
Db2 ODBC
advanced features 411
advantages of using 7
application requirements 44
common problems 500
components 44
diagnostic trace 510
functions 79
initialization file 54
installing 37
limited block fetch 434
restrictions on ODBC 11
shadow catalog 423
stored procedures 466
traces, debugging 507
DB2 ODBC
configuring 37
encapsulation 7
improving performance 501
performance 7
security 7
DBRMs, binding 39
deadlocks
application
deadlocks 478
debugging 519
DECIMAL
conversion to C 549
length 544
precision 542
scale 543
deferred arguments 17
DELETE 20
delete from rowset
SQLSetPos() 450
deprecated functions
list of 562
SQLAllocConnect() 86
SQLAllocEnv() 87
SQLAllocStmt() 92
SQLColAttributes() 137
SQLFreeConnect() 213
SQLFreeEnv() 214
SQLGetConnectOption() 223
SQLGetStmtOption() 291
SQLSetConnectOption() 363
deprecated functions *(continued)*
SQLSetParam() 371
SQLSetStmtOption() 389
SQLTransact() 410
diagnosis
description 23
function, SQLGetDiagRec() 239
trace 510
diagnostic trace
activating in z/OS 510
activating in z/OS UNIX 511
diagnostic trace command 514
diagnostic trace file 515
diagnostic trace, obtaining 518
disability x
DISCONNECT, plan bind option 42
disconnecting 165
display size of SQL data types 545
display size, data type attributes 542
distinct types
defining 465
distributed transactions 414
distributed unit of work 414
DOUBLE
conversion to C 549
display size 545
length 544
precision 542
scale 543
driver
DB2 ODBC 521
ODBC 521
ODBC 3.0 behavior 565
driver manager 521
DSN8O3VP, JCL sample 46, 568
DSN8OIVP, JCL sample 46
DSNAO64T 514
DSNAOINI data definition 54
DSNAOTRC 514
DSNAOTRX 514
DSNTEJ8, application sample 46
DSNTIJCL, bind sample 39, 42
dynamic SQL
statement caching 503
dynamic statement cache 503
DYNAMICRULES, bind option for ODBC applications 41

## E

EBCDIC 479
embedded SQL
comparison to Db2 ODBC 4
mixing with Db2 ODBC 362, 493
encoding schemes
guidelines 479
entry points 479
environment
attributes
description 411, 413
SQLSetEnvAttr() 367
handle
SQLAllocHandle() 87
SQLFreeHandle() 215
handles

environment *(continued)*
handles *(continued)*
allocating 10
description 4
freeing 10
information, querying 35
run time 37
z/OS UNIX
export statements 54
setup 43
variables 54
error code, internal 520
errors, extended scalar functions 525
errors, retrieving information about 175
escape character 498
escape clause syntax 495
escape clauses, vendor
SQLNativeSql() 306
example
scrollable cursor 440
examples
application trace output 508
array INSERT 424
binding decimal floating point data 454
binding UTF-8 data 485
binding variable-length timestamp data 455
catalog functions 422
column-wise binding 433
converting C to SQL data 561
converting SQL to C data 555
distinct types 465
large objects 460
mixing embedded SQL and Db2 ODBC 493
retrieve UCS-2 data 485
retrieving UTF-8 data 486
row-wise binding 433
single Language Environment thread 475
using suffix-W APIs 487
XML column updates 461
execute, application 46, 53
executing a statement 16
executing directly 16
executing SQL
SQLExecDirect() 177
SQLExecute() 183
SQLPrepare() 317
export statements 54
extended fetch 186
extended indicators 500
extended SQL syntax 496
Extra Performance Linkage 39

## F

fetching
scrollable cursor 438
SQLExtendedFetch() 186
SQLFetch() 192
SQLFetchScroll() 198
SQLSetPos() 372
FLOAT
conversion to C 549
display size 545
length 544
FLOAT *(continued)*
precision 542
scale 543
foreign key columns, list 206
freeing handles 215
function, date and time 526
functions overview 80
functions, complete list 80
functions, DB2 ODBC 79
functions, string 525
functions, support for ODBC 522
functions, system 526

## G

general-use programming information, described 590
generic API 479, 480
get functions
SQLGetConnectAttr() 220
SQLGetCursorName() 223
SQLGetData() 228
SQLGetDiagRec() 239
SQLGetEnvAttr() 242
SQLGetFunctions() 244
SQLGetPosition() 278
SQLGetSQLCA() 283
SQLGetStmtAttr() 287
SQLGetSubString() 291
SQLGetTypeInfo() 295
global transactions 418
GRAPHIC
conversion to C 548

## H

handles
connection 4, 10
environment 4, 10
SQLAllocHandle() 87
SQLFreeHandle() 215
statement 4
handles, function arguments 30
HOUR scalar function 526

## I

IFNULL scalar function 526
improving performance 501
index information, retrieving 396
initialization
file
common errors 55, 500
defaults, changing 411
description 54
specifying 54
structure of 55
tasks 9, 10
initialization keywords
ACCOUNTINGINTERVAL 56
APPLTRACE 56
APPLTRACEFILENAME 56
AUTOCOMMIT 56
BITDATA 56

initialization keywords *(continued)*
CLISCHEMA 56
COLLECTIONID 56
CONNECTTYPE 56
CURRENTAPPENSCH 56
CURRENTSQLID 56
CURSORHOLD 56
DB2EXPLAIN 56
DBNAME 56
DIAGTRACE 56
DIAGTRACE_BUFFER_SIZE 56
DIAGTRACE_NO_WRAP 56
EXTENDEDTABLEINFO 56
GRANTEELIST 56
GRANTORLIST 56
GRAPHIC 56
LIMITEDBLOCKFETCH 56
MAXCONN 56
MULTICONTEXT 56
MVSATTACHTYPE 56
MVSDEFAULTSSID 56
OPTIMIZEFORNROWS 56
PATCH2 56
PLANNAME 56
QUERYDATASIZE 56
REPORTPUBLICPRIVILEGES 56
RETCATALOGCURRSERVER 56
RETURNALIASES 56
SCHEMALIST 56
STREAMBUFFERSIZE 56
SYSSCHEMA 56
TABLETYPE 56
THREADSAFE 56
TXNISOLATION 56
UNDERSCORE 56
INSERT 20
INSERT scalar function 525
inserts
SQLBulkOperations() 121
inserts, bulk
SQLBulkOperations() 447, 453
installation 37
INTEGER
conversion to C 549
display size 545
length 544
precision 542
scale 543
internal error code 520
introduction to Db2 ODBC 1
INVALID_HANDLE 23
isolation levels, concurrency and consistency 501
isolation levels, ODBC 524

## K

keywords, initialization 55, 56

## L

Language Environment threads
multiple 473
multiple contexts 475
large objects
example 460
file reference variables 460
locators
finding the position of a string 278
retrieving a portion of a string 291
retrieving string length 276
manipulating large data values 505
LEFT scalar function 525
length of SQL data types 544
LENGTH scalar function 525
length, data type attributes 542
LIKE predicate 498
limited block fetch
Db2 ODBC 434
link-edit, application 50
link-editing
64-bit applications 52
XPLINK applications 51
links
non-IBM Web sites 590
LOB 458
LOB locators
example 460
get functions
SQLGetLength() 276
SQLGetLength() 276
SQLGetPosition() 278
SQLGetSubString() 291
using 458
long bulk inserts
SQLBulkOperations() 453
long data
retrieving in pieces 451
sending in pieces
SQLParamData() 313
SQLPutData() 340
LONGVARBINARY 550
LONGVARCHAR
conversion to C 547
DECIMAL
display size 545
display size 545
length 544
precision 542
scale 543
LONGVARGRAPHIC 548

## M

manual-commit mode 21
MERGE 20
metadata character 421
migrating ODBC applications
to 64-bit 76
migrating to Db2 11
ODBC driver 75
MINUTE scalar function 526
mixed applications 362
MONTH scalar function 526
multiple contexts
multiple Language Environment threads 475
single Language Environment thread 475
multithreaded applications 471, 473

## N


name, cursor 364
native error code 24
native SQL 306
network flow, reducing 504, 505
NOW scalar function 526
NOXPLINK, compiling 47
nul-terminated strings 34
NUMERIC
  conversion to C 549
  display size 545
  length 544
  precision 542
  scale 543


## O


ODBC
  and Db2 ODBC 1, 521
  APIs and data types 522
  connectivity 11
  core level functions 1
  date data 496
  error reporting 24
  function list 80
  functions overview 80
  isolation levels 524
  level conformance 2
  SQL extensions 496
  support for ODBC features 2
  time data 496
  timestamp data 496
  vendor escape clauses 496
ODBC 3.0 driver behavior 565
ODBC driver
  64-bit 39
  migrating to Db2 11 75
outer join syntax 497


## P


packages, binding 39
parameter markers
  binding 17
  casting to distinct types 500
  casting to source types 500
  decimal floating point data 454
  introduction 4
  multiple result sets 303
  number in a statement 309
  passing data to 340
  retrieving a description of 162
  sending data in pieces 313
  specifying an input array 315
  using arrays 424
  variable-length timestamp data 455
parameters, stored procedure 327
passwords 149
pattern value 421
plan, binding 42
pointers, function arguments 30
portability
portability *(continued)*
  techniques for maximizing 505
precision of SQL data types 542
precision, data type attributes 542
prelink, application 50
prepare SQL statements 317
preparing a statement 16
primary key
  SQLPrimaryKeys() 323
  SQLSpecialColumns() 390
private external contexts 477
privileges
  SQLColumnPrivileges() 138
  SQLTablePrivileges() 401
programming interface information, described 590


## Q


query statements, processing 18
querying
  catalog information 420
  data source information 35
  Db2 catalog 420
  environment information 35
  shadow catalog 423


## R


REAL
  conversion to C 549
  display size 545
  length 544
  precision 542
  scale 543
reducing network flow 504, 505
remote site, creating packages at 40
REOPT, bind option for ODBC applications 41
REPEAT scalar function 525
result sets
  function generated 505
  retrieving into array 428
  returning from stored procedures 468
  SQLExtendedFetch() 186
  SQLFetch() 192
  SQLFetchScroll() 198
  SQLGetData() 228
  SQLMoreResults() 303
  SQLNumResultCols() 310
  SQLSetPos() 372
retrieving data
  scrollable cursor 438
retrieving data, guidelines 502
retrieving multiple rows 428
return codes 23
RIGHT scalar function 525
rollback 21, 172
row identifiers 390
row set 186
row status array
  ODBC 432
row-wise binding 429, 430
ROWID
  conversion to C 553

ROWID *(continued)*
display size 545
length 544
precision 542
scale 543
rows, number changed 343
rowset
row status array 432
rowset, ODBC
description 436
retrieval 436
run time environment
setting up 39
support 37

## S

samples
APD29 572
DSN8O3VP 46, 568
DSN8OIVP 46
DSNTEJ8 46
DSNTIJCL 39, 42
examples
stored procedure 568
scalar functions 499
scale of SQL data types 543
scale, data type attributes 542
scrollable cursor
example 440
ODBC 435, 438
scrollable cursor, advantages 435
search argument 421
SECOND scalar function 526
SELECT 18
service trace 507
set functions
SQLSetColAttributes() 346
SQLSetConnectAttr() 350
SQLSetConnection() 362
SQLSetCursorName() 364
SQLSetEnvAttr() 367
SQLSetStmtAttr() 378
shadow catalog
querying 423
shortcut keys
keyboard x
SMALLINT
conversion to C 549
display size 545
length 544
precision 542
scale 543
SMP/E jobs 37
SQL
data type attributes 542
dynamically prepared 4
parameter markers 17
preparing and executing statements 16
query statements 18
SELECT 18
VALUES 18
SQL Access Group 1
SQL_ATTR_ACCESS_MODE 350
SQL_ATTR_AUTOCOMMIT 350, 505
SQL_ATTR_BIND_TYPE 378
SQL_ATTR_CLOSE_BEHAVIOR 378
SQL_ATTR_CONCURRENCY 378
SQL_ATTR_CONNECTTYPE
distributed unit of work 414
SQLGetEnvAttr() 367
SQLSetConnectAttr() 350
SQL_ATTR_CURRENT_SCHEMA 350
SQL_ATTR_CURSOR_HOLD 378, 502
SQL_ATTR_CURSOR_TYPE 378
SQL_ATTR_DB2EXPLAIN 350
SQL_ATTR_MAX_LENGTH 378
SQL_ATTR_MAX_ROWS 378, 500
SQL_ATTR_MAXCONN 350, 367
SQL_ATTR_NODESCRIBE 378
SQL_ATTR_NOSCAN 378, 504
SQL_ATTR_ODBC_VERSION 367
SQL_ATTR_OUTPUT_NTS 367
SQL_ATTR_PARAMOPT_ATOMIC 350
SQL_ATTR_RETRIEVE_DATA 378
SQL_ATTR_ROW_ARRAY_SIZE 378, 429
SQL_ATTR_ROWSET_SIZE 378, 429
SQL_ATTR_STMTTXN_ISOLATION 378
SQL_ATTR_SYNC_POINT 350
SQL_ATTR_TXN_ISOLATION
setting isolation levels 501
SQLSetConnectAttr() 350
SQLSetStmtAttr() 378
SQL_BIGINT 25
SQL_BINARY 25
SQL_BLOB 25
SQL_BLOB_LOCATOR 25
SQL_C_BIGINT 25, 557
SQL_C_BINARY 25, 558
SQL_C_BINARYXML 558
SQL_C_BIT 25, 557
SQL_C_BLOB_LOCATOR 25
SQL_C_CHAR 25, 556
SQL_C_CLOB_LOCATOR 25
SQL_C_DBCHAR 25, 559
SQL_C_DBCLOB_LOCATOR 25
SQL_C_DEFAULT 26
SQL_C_DOUBLE 25, 557
SQL_C_FLOAT 25, 557
SQL_C_LONG 25, 557
SQL_C_SHORT 25, 557
SQL_C_TINYINT 25, 557
SQL_C_TYPE_DATE 25, 559
SQL_C_TYPE_TIME 25, 560
SQL_C_TYPE_TIMESTAMP 25, 560
SQL_C_TYPE_TIMESTAMP_STRUCT_EXT 25
SQL_C_WCHAR 25
SQL_CCSID_CHAR 378
SQL_CCSID_GRAPHIC 378
SQL_CHAR 25
SQL_CLOB 25
SQL_CLOB_LOCATOR 25
SQL_CONCURRENT_TRANS 414
SQL_COORDINATED_TRANS 414
SQL_DATA_AT_EXEC 451
SQL_DBCLOB 25
SQL_DBCLOB_LOCATOR 25
SQL_DECFLOAT 25

SQL_DECIMAL 25
SQL_DOUBLE 25
SQL_ERROR 23
SQL_FLOAT 25
SQL_GRAPHIC 25
SQL_INTEGER 25
SQL_LONGVARBINARY 25
SQL_LONGVARCHAR 25
SQL_LONGVARGRAPHIC 25
SQL_NEED_DATA 23
SQL_NO_DATA_FOUND 23
SQL_NTS 34
SQL_NUMERIC 25
SQL_REAL 25
SQL_ROWID 25
SQL_SMALLINT 25
SQL_SUCCESS 23
SQL_SUCCESS_WITH_INFO 23
SQL_TYPE_DATE 25
SQL_TYPE_TIME 25
SQL_TYPE_TIMESTAMP 25
SQL_VARBINARY 25
SQL_VARCHAR 25
SQL_VARGRAPHIC 25
SQL_XML 25
SQL, native 306
SQLAllocConnect(), deprecated function 86
SQLAllocEnv(), deprecated function 87
SQLAllocHandle()
  description 87
  introduction 14
SQLAllocStmt(), deprecated function 92
SQLBindCol()
  description 92
  introduction 14
  processing query statements 18
SQLBindFileToCol()
  description 99
SQLBindFileToParam()
  description 103
SQLBindParameter()
  data conversion example 31
  description 106
  introduction 14
  processing query statements 18
  using 16, 17
SQLBulkOperations()
  bulk inserts 447
  description 121
  long data for bulk inserts 453
SQLCA data structure 283
SQLCancel()
  description 125
  use in data-at-execute 451
SQLCHAR 25
SQLCloseCursor() 127
SQLColAttribute()
  description 129
  introduction 14
  processing query statements 18
SQLColAttributes(), deprecated function 137
SQLColumnPrivileges() 138
SQLColumns() 142
SQLConnect() 149
SQLDataSources()
  description 154
  introduction 14
SQLDBCHAR 25
SQLDescribeCol()
  description 157
  introduction 14
  processing query statements 18
SQLDescribeParam()
  using 17
SQLDisconnect() 165
SQLDOUBLE 25
SQLDriverConnect()
  connection string 411
  description 167
  instead of SQLConnect() 505
SQLEndTran()
  description 172
  effects of calling 22
  introduction 14
  processing query statements 18
  using 21
  when to call 21
SQLERROR, bind option for ODBC applications 41
SQLError() 175
SQLExecDirect()
  description 177
  introduction 14
  using 16
SQLExecute()
  description 183
  introduction 14
  SQLPrepare()
    using 16
  using 16
SQLExtendedFetch() 186
SQLFetch()
  description 192
  introduction 14
  processing query statements 18
SQLFetchScroll()
  description 198
SQLForeignKeys() 206
SQLFreeConnect(), deprecated function 213
SQLFreeEnv(), deprecated function 214
SQLFreeHandle()
  description 215
  freeing statement handles 22
  introduction 14
SQLFreeStmt() 217
SQLGetConnectAttr() 220
SQLGetConnectOption(), deprecated function 223
SQLGetCursorName()
  retrieving name of cursor 20
SQLGetData()
  description 228
  introduction 14
  processing query statements 18
SQLGetDiagRec() 239
SQLGetEnvAttr() 242
SQLGetFunctions() 244
SQLGetInfo()
  changes to InfoType values 563
SQLGetLength() 276

SQLGetPosition() 278
SQLGetSQLCA()
  retrieving SQLCA 25
SQLGetStmtAttr() 287
SQLGetStmtOption() 291
SQLGetSubString() 291
SQLGetTypeInfo() 295
SQLHDBC 30, 31
SQLHENV 30, 31
SQLHSTMT 30, 31
SQLINTEGER 25
SQLMoreResults()
  description 303
  using 424
SQLNativeSql() 306
SQLNumParams()
  using 17
SQLNumResultCols()
  description 310
  introduction 14
  processing query statements 18
SQLParamData()
  description 313
  use in data-at-execute 451
SQLParamOptions() 315
SQLPOINTER 30, 31
SQLPrepare()
  description 317
  introduction 14
SQLPrimaryKeys() 323
SQLProcedureColumns() 327
SQLProcedures() 336
SQLPutData()
  description 340
  use in data-at-execute 451
SQLREAL 25
SQLRETURN 30, 31
SQLRowCount()
  description 343
  introduction 14
SQLSCHAR 25
SQLSetColAttributes()
  reducing network flow 504
SQLSetConnectAttr()
  changes to attributes 563
SQLSetConnection() 362
SQLSetConnectOption(), deprecated function 363
SQLSetCursorName()
  string arguments 35
SQLSetEnvAttr()
  changes to attributes 564
SQLSetParam(), deprecated function 371
SQLSetPos()
  delete from rowset 450
  description 372
  update rowset 448
SQLSetStmtAttr()
  changes to attributes 564
SQLSetStmtOption(), deprecated function 389
SQLSMALLINT 25
SQLSpecialColumns() 390
SQLSTATE
  description 24
  format of 24
SQLSTATE *(continued)*
  function cross reference 527
  in Db2 ODBC 4
  mappings 565
SQLStatistics() 396
SQLTablePrivileges() 401
SQLTables() 405
SQLTransact(), deprecated function 410
SQLUINTEGER 30, 31
SQLUSMALLINT 30, 31
SQLWCHAR 25, 30, 31
SRRBACK service 418
SRRCMIT service 418
statement
  attributes
    description 413
statements
  attributes
    description 411
    setting 378
    SQLGetStmtAttr() 287
  handles
    allocating 15
    freeing 22
    introduction 4
    maximum number of 15
    SQLAllocHandle() 87
    SQLFreeHandle() 215
    SQLFreeStmt() 217
  preparing 317
statistics, retrieving 396
stored procedures
  advantages 467
  binding 40
  calls in Db2 ODBC 467
  diagnostic trace, obtaining 517
  example 568
  functions
    SQLProcedureColumns() 327
    SQLProcedures() 336
  obtaining an application trace 517
  ODBC escape clause 498
  receive result sets 469
  restrictions 469
  return result sets 468
  returning result sets 303, 468
  tracing 516
  using with Db2 ODBC 466
  writing in Db2 ODBC 468
string
  arguments 34, 35
  nul-termination 34
  truncation 35
string function 525
SUBSTRING scalar function 525
subsystem, defining 54
Suffix-W APIs 481
syntax diagram
  how to read xi
system, functions 526

## T

table information, retrieving 405

tables, list of privileges 401
termination tasks 9, 10
threads 471, 473
TIME
  conversion to C 551
  display size 545
  length 544
  precision 542
  scale 543
TIME_STRUCT 25
timeouts 478
TIMESTAMP
  conversion to C 552
  display size 545
  length 544
  precision 542
  scale 543
TIMESTAMP WITH TIMEZONE
  display size 545
  length 544
  scale 543
TIMESTAMP_STRUCT 25
TIMESTAMP_STRUCT_EXT 25
trace
  a client application 516
  application 507
  Db2 ODBC diagnosis 510
  stored procedure 516
trace file name, formats 507
trace types 507
tracing a client application 516
transaction
  isolation levels, ODBC 524
  management 21
  processing 9
transactions, ending 172
truncation 35

## U

Unicode 479
unique indexes 390
unit of work, semantics 416
UPDATE 20
update rowset
  SQLSetPos() 448
user IDs 149
USER scalar function 526

## V

VALUES 18
VARBINARY
  length 544
  precision 542
  scale 543
  XML
    length 544
VARCHAR
  conversion to C 547
  display size 545
  length 544
  precision 542
VARCHAR *(continued)*
  scale 543
VARGRAPHIC 548
vendor escape clauses
  disabling 504
  function for determining 495
  SQLNativeSql() 306
  using 495

## W

writing Db2 ODBC applications 9

## X

X/Open CAE 24
X/Open Company 1
X/Open SQL CLI 1
XML
  column updates 461
  conversion to C 554
  data retrieval 463
  data types 461
  display size 545
  precision 542
  scale 543
XPLINK 39
XPLINK applications
  link-editing 51
XPLINK ODBC driver 39
XPLINK, compiling 48

## Z

z/OS Unix
  diagnostic trace file 515
z/OS UNIX
  activating diagnostic trace 511
  compile application 47
  DSNTEJ8 50
  DSNTEJ8X 50
  environment setup 43
  environment variable 54
  execute application 53
  export statements 54
  prelink and link-edit 50



Product Number: 5615-DB2
5697-P43

SC19-4063-05

