

DB2 Universal Database for OS/390 and z/OS



ODBC Guide and Reference

Version 7

DB2 Universal Database for OS/390 and z/OS



ODBC Guide and Reference

Version 7

Note

Before using this information and the product it supports, be sure to read the general information under Appendix G, "Notices", on page 535.

Fourth Edition, Softcopy Only (June 2003)

This edition applies to Version 7 of IBM DATABASE 2 Universal Database Server for OS/390 and z/OS (DB2 for OS/390 and z/OS), 5675-DB2, and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

This softcopy version is based on the printed edition of the book and includes the changes indicated in the printed version by vertical bars. Additional changes made to this softcopy version of the book since the hardcopy book was published are indicated by the hash (#) symbol in the left-hand margin. Editorial changes that have no technical significance are not noted.

This and other books in the DB2 for OS/390 and z/OS library are periodically updated with technical changes. These updates are made available to licensees of the product on CD-ROM and on the Web (currently at www.ibm.com/software/data/db2/os390/library.html). Check these resources to ensure that you are using the most current information.

© Copyright International Business Machines Corporation 1997, 2001. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book	ix
Who should use this book	ix
Product terminology and citations	ix
How to send your comments	x

#

Summary of changes to this book	xi
--	----

Chapter 1. Summary of changes to DB2 for OS/390 and z/OS Version 7.	1
Enhancements for managing data	1
Enhancements for reliability, scalability, and availability.	1
Easier development and integration of e-business applications.	3
Improved connectivity	4
Features of DB2 for OS/390 and z/OS.	4
Migration considerations	4

Chapter 2. Introduction to DB2 ODBC	7
DB2 ODBC background information.	7
Differences between DB2 ODBC and ODBC version 2.0	7
Differences between DB2 ODBC and embedded SQL	9
Advantages of using DB2 ODBC	11
Deciding which interface to use	12
Static and dynamic SQL	12
Use both interfaces	13
Write a mixed application	13
Other information sources	13

Chapter 3. Writing a DB2 ODBC application	15
Initialization and termination	16
Handles	16
ODBC connection model	17
Connect type 1 and type 2	18
Connecting to one or more data sources	18
Transaction processing	20
Allocating statement handles	22
Preparation and execution.	22
Processing results.	24
Commit or rollback	26
Freeing statement handles	28
Diagnostics	28
Function return codes	28
SQLSTATEs	29
SQLCA.	30
Data types and data conversion	30
C and SQL data types	30
Other C data types	33
Data conversion	34
Working with string arguments	35
Length of string arguments	35
Null-termination of strings	36
String truncation	36
Interpretation of strings	37
Querying environment and data source information	37
Querying environment information example	38

#

Chapter 4. Configuring DB2 ODBC and running sample applications	41
Installing DB2 ODBC	41
DB2 ODBC runtime environment	41
Connectivity requirements	42
Setting up DB2 ODBC runtime environment	43
Bind DBRMs to packages	43
Bind packages at remote sites	45
Bind stored procedures	45
Bind an application plan	45
Setting up OS/390 UNIX environment	46
Setting up suffix-W API support	46
Preparing and executing a DB2 ODBC application	48
DB2 ODBC application requirements	49
Application preparation and execution steps	50
DB2 ODBC initialization file	52
Using the initialization file	53
Initialization keywords	55
DB2 ODBC migration considerations	65
Chapter 5. Functions	67
Function summary	68
SQLAllocConnect - Allocate connection handle	73
SQLAllocEnv - Allocate environment handle	77
SQLAllocHandle - Allocate handle	79
SQLAllocStmt - Allocate a statement handle	83
SQLBindCol - Bind a column to an application variable	85
SQLBindParameter - Binds a parameter marker to a buffer or LOB locator	91
SQLCancel - Cancel statement	102
SQLCloseCursor - Close cursor and discard pending results	104
SQLColAttribute - Get column attributes	106
SQLColAttributes - Get column attributes	114
SQLColumnPrivileges - Get privileges associated with the columns of a table	120
SQLColumns - Get column information for a table	124
SQLConnect - Connect to a data source	129
SQLDataSources - Get list of data sources	134
SQLDescribeCol - Describe column attributes	137
SQLDescribeParam - Describe parameter marker	142
SQLDisconnect - Disconnect from a data source	144
SQLDriverConnect - (Expanded) connect to a data source	146
SQLEndTran - End transaction of a connection	152
SQLError - Retrieve error information	155
SQLExecDirect - Execute a statement directly	161
SQLExecute - Execute a statement	166
SQLExtendedFetch - Extended fetch (fetch array of rows)	169
SQLFetch - Fetch next row	176
SQLForeignKeys - Get the list of foreign key columns	181
SQLFreeConnect - Free connection handle	189
SQLFreeEnv - Free environment handle	191
SQLFreeHandle - Free handle resources	193
SQLFreeStmt - Free (or reset) a statement handle	196
SQLGetConnectAttr - Get current attribute setting	199
SQLGetConnectOption - Returns current setting of a connect option	202
SQLGetCursorName - Get cursor name	204
SQLGetData - Get data from a column	210
SQLGetDiagRec - Get multiple field settings of diagnostic record	223
SQLGetEnvAttr - Returns current setting of an environment attribute	226

SQLGetFunctions - Get functions.	228
SQLGetInfo - Get general information	234
SQLGetLength - Retrieve length of a string value.	257
SQLGetPosition - Return starting position of string	259
SQLGetSQLCA - Get SQLCA data structure	263
SQLGetStmtAttr - Get current setting of a statement attribute	270
SQLGetStmtOption - Returns current setting of a statement option	273
SQLGetSubString - Retrieve portion of a string value	275
SQLGetTypeInfo - Get data type information	278
SQLMoreResults - Determine if there are more result sets	286
SQLNativeSql - Get native SQL text	290
SQLNumParams - Get number of parameters in a SQL statement	292
SQLNumResultCols - Get number of result columns.	294
SQLParamData - Get next parameter for which a data value is needed	296
SQLParamOptions - Specify an input array for a parameter	298
SQLPrepare - Prepare a statement	300
SQLPrimaryKeys - Get primary key columns of a table.	308
SQLProcedureColumns - Get input/output parameter information for a procedure	313
SQLProcedures - Get list of procedure names	323
SQLPutData - Passing data value for a parameter	327
SQLRowCount - Get row count	330
SQLSetColAttributes - Set column attributes	332
SQLSetConnectAttr - Set connection attributes.	336
SQLSetConnection - Set connection handle.	343
SQLSetConnectOption - Set connection option.	345
SQLSetCursorName - Set cursor name	347
SQLSetEnvAttr - Set environment attribute	350
SQLSetParam - Binds a parameter marker to a buffer	354
SQLSetStmtAttr - Set options related to a statement	360
SQLSetStmtOption - Set statement option	367
SQLSpecialColumns - Get special (row identifier) columns	369
SQLStatistics - Get index and statistics information for a base table	374
SQLTablePrivileges - Get privileges associated with a table	379
SQLTables - Get table information	382
SQLTransact - Transaction management	386
Chapter 6. Using advanced features.	389
Environment, connection, and statement options	389
Distributed unit of work (coordinated distributed transactions)	391
Options that govern distributed unit of work semantics	392
Establishing a coordinated transaction connection	394
Global transaction processing	396
Querying system catalog information	397
Using the catalog query functions	397
Directing catalog queries to the DB2 ODBC shadow catalog.	399
Sending/retrieving long data in pieces	401
Specifying parameter values at execute time	401
Fetching data in pieces	402
Using arrays to input parameter values	403
Array input example	405
Retrieving a result set into an array	406
Returning array data for column-wise bound data.	407
Returning array data for row-wise bound data	408
Column-wise, row-wise binding example	409
Using large objects	411

Using LOB locators	412
Using distinct types	414
Distinct type example	415
Using stored procedures	417
Advantages of using stored procedures	417
Catalog table for stored procedures	418
Calling stored procedures from a DB2 ODBC application	418
Writing a DB2 ODBC stored procedure	419
Returning result sets from stored procedures	420
Writing multithreaded applications	421
DB2 ODBC support of multiple LE threads	421
When to use multiple LE threads	423
DB2 ODBC support of multiple contexts	424
Application deadlocks	428
Using Unicode functions	429
Background	429
DB2 ODBC Unicode support	429
Application programming guidelines	433
Example: ODBC application using suffix-W APIs	434
Mixing embedded SQL and DB2 ODBC	446
Mixed embedded SQL and DB2 ODBC example	447
Using vendor escape clauses	448
Escape clause syntax	449
Using ODBC defined SQL extensions	449
ODBC date, time, timestamp data	449
ODBC outer join syntax	450
Like predicate escape clauses	450
Stored procedure CALL syntax	451
ODBC scalar functions	451
Programming hints and tips	452
Avoiding common initialization file problems	452
Setting common connection options	452
Setting common statement options	452
Using SQLSetColAttributes to reduce network flow	453
Comparing binding and SQLGetData	453
Increasing transfer efficiency	454
Limiting use of catalog functions	454
Using column names of function generated result sets	454
Making use of dynamic SQL statement caching	454
Optimizing insertion and retrieval of data	455
Optimizing for large object data	455
Using SQLDriverConnect instead of SQLConnect	455
Turning off statement scanning	455
Casting distinct types	455
Chapter 7. Problem diagnosis	457
Tracing	457
Application trace	457
Diagnostic trace	459
Stored procedure trace	462
Debugging	466
Abnormal termination	466
Internal error code	466
Appendix A. DB2 ODBC and ODBC	467
DB2 ODBC and ODBC drivers	467

ODBC APIs and data types	468
Isolation levels	470
Appendix B. Extended scalar functions	471
String functions	471
Date and time functions	472
System functions.	472
Appendix C. SQLSTATE cross reference	475
Appendix D. Data conversion	485
Data type attributes.	485
Precision	485
Scale	486
Length	487
Display size	488
Converting data from SQL to C data types	489
Converting character SQL data to C data.	490
Converting graphic SQL data to C data	491
Converting numeric SQL data to C data	492
Converting binary SQL data to C data	493
Converting date SQL data to C data	493
Converting time SQL data to C data	493
Converting timestamp SQL data to C data	494
Converting row ID SQL data to C data.	495
SQL to C data conversion examples	495
Converting data from C to SQL data types	495
Converting character C data to SQL data.	496
Converting numeric C data to SQL data	497
Converting binary C data to SQL data	498
Converting DBCHAR C data to SQL data.	498
Converting date C data to SQL data	498
Converting time C data to SQL data	499
Converting timestamp C data to SQL data	499
C to SQL data conversion examples	500
Appendix E. Deprecated function	501
Mapping deprecated functions.	501
Changes to SQLGetInfo information types	501
Changes to SQLSetConnectAttr attributes	502
Changes to SQLSetEnvAttr attributes	502
Changes to SQLSetStmtAttr attributes	502
ODBC 3.0 driver behavior	503
SQLSTATE mappings	504
Changes to datetime data types	505
Appendix F. Example code	507
DSN803VP sample application	508
Client application calling a DB2 ODBC stored procedure	514
Appendix G. Notices	535
Programming interface information	536
Trademarks.	536
Glossary	539

I

Bibliography	545
Index	553

About this book

This book provides the information necessary to write applications using DB2[®] ODBC to access IBM[®] DATABASE 2[™] servers, as well as any database that supports DRDA[®] level 1 or DRDA level 2 protocols. This book should also be used as a supplement when writing portable ODBC applications that can be executed in a native DB2 for OS/390[®] and z/OS[™] environment using DB2 ODBC.

Who should use this book

This book is intended for:

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

Important

In this version of DB2 for OS/390 and z/OS, some utility functions are available as optional products. 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.

Product terminology and citations

In this book, DB2 Universal Database[™] Server for OS/390 and z/OS is referred to as "DB2 for OS/390 and z/OS." In cases where the context makes the meaning clear, DB2 for OS/390 and z/OS is referred to as "DB2." When this book refers to other books in this library, a short title is used. (For example, "See *DB2 SQL Reference*" is a citation to *IBM DATABASE 2 Universal Database Server for OS/390 and z/OS SQL Reference*.)

When referring to a DB2 product other than DB2 for OS/390 and z/OS, this book uses the product's full name to avoid ambiguity.

The following terms are used as indicated:

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

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

CICS[®] Represents CICS/ESA[®] and CICS Transaction Server for OS/390.

IMS[™] Represents IMS or IMS/ESA[®].

MVS Represents the MVS element of OS/390.

OS/390
Represents the OS/390 or z/OS operating system.

RACF[®]
Represents the functions that are provided by the RACF component of the SecureWay[®] Security Server for OS/390 or by the RACF component of the OS/390 Security Server.

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 OS/390 and z/OS documentation.

You can use any of the following methods to provide comments:

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

<http://www.ibm.com/software/db2os390>

The Web site has a feedback page that you can use to send comments.

- Complete the readers' comment form at the back of the book and return it by mail, by fax (800-426-7773 for the United States and Canada), or by giving it to an IBM representative.

Summary of changes to this book

The major changes to this edition of the book include:

- # • Instructions for installing and configuring the conversion environment required for
Unicode UCS-2 support. See “Setting up suffix-W API support” on page 46.
- # • User ID and password support for authenticating the end user on a connection
string to a designated server. See “SQLConnect - Connect to a data source” on
page 129 and “SQLDriverConnect - (Expanded) connect to a data source” on
page 146.

Chapter 1. Summary of changes to DB2 for OS/390 and z/OS Version 7

DB2 for OS/390 and z/OS Version 7 delivers an enhanced relational database server solution for OS/390. This release focuses on greater ease and flexibility in managing your data, better reliability, scalability, and availability, and better integration with the DB2 family.

In Version 7, some utility functions are available as optional products; you must separately order and purchase a license to such utilities. Discussion of utility functions in this publication is not intended to otherwise imply that you have a license to them. See *DB2 Utility Guide and Reference* for more information about utilities products.

Enhancements for managing data

Version 7 delivers the following enhancements for managing data:

- DB2 now collects a comprehensive statistics history that:
 - Lets you track changes to the physical design of DB2 objects
 - Lets DB2 predict future space requirements for table spaces and indexes more accurately and run utilities to improve performance
- Database administrators can now manage DB2 objects more easily and no longer must maintain their utility jobs (even when new objects are added) by using enhancements that let them:
 - Dynamically create object lists from a pattern-matching expression
 - Dynamically allocate the data sets that are required to process those objects
- More flexible DBADM authority lets database administrators create views for other users.
- Enhancements to management of constraints let you specify a constraint at the time you create primary or unique keys. A new restriction on the DROP INDEX statement requires that you drop the primary key, unique key, or referential constraint before you drop the index that enforces a constraint.

Enhancements for reliability, scalability, and availability

Version 7 delivers the following enhancements for the reliability, scalability, and availability of your e-business:

- The DB2 Utilities Suite provides utilities for all of your data management tasks that are associated with the DB2 catalog.
- The new UNLOAD utility lets you unload data from a table space or an image copy data set. In most cases, the UNLOAD utility is faster than the DSNTIAUL sample program, especially when you activate partition parallelism for a large partitioned table space. UNLOAD is also easier to use than REORG UNLOAD EXTERNAL.
- The new COPYTOCOPY utility lets you make additional image copies from a primary image copy and registers those copies in the DB2 catalog. COPYTOCOPY leaves the target object in read/write access mode (UTRW), which allows Structured Query Language (SQL) statements and some utilities to run concurrently with the same target objects.

- Parallel LOAD with multiple inputs lets you easily load large amounts of data into partitioned table spaces for use in data warehouse applications or business intelligence applications. Parallel LOAD with multiple inputs runs in a single step, rather than in different jobs.
- A faster online REORG is achieved through the following enhancements:
 - Online REORG no longer renames data sets, which greatly reduces the time that data is unavailable during the SWITCH phase.
 - Additional parallel processing improves the elapsed time of the BUILD2 phase of REORG SHRLEVEL(CHANGE) or SHRLEVEL(REFERENCE).
- More concurrency with online LOAD RESUME is achieved by letting you give users read and write access to the data during LOAD processing so that you can load data concurrently with user transactions.
- More efficient processing for SQL queries:
 - More transformations of subqueries into a join for some UPDATE and DELETE statements
 - Fewer sort operations for queries that have an ORDER BY clause and WHERE clauses with predicates of the form *COL=constant*
 - More parallelism for IN-list index access, which can improve performance for queries involving IN-list index access
- The ability to change system parameters without stopping DB2 supports online transaction processing and e-business without interruption.
- Improved availability of user objects that are associated with failed or canceled transactions:
 - You can cancel a thread without performing rollback processing.
 - Some restrictions imposed by the restart function have been removed.
 - A NOBACKOUT option has been added to the CANCEL THREAD command.
- Improved availability of the DB2 subsystem when a log-read failure occurs: DB2 now provides a timely warning about failed log-read requests and the ability to retry the log read so that you can take corrective action and avoid a DB2 outage.
- Improved availability in the data sharing environment:
 - Group attachment enhancements let DB2 applications generically attach to a DB2 data sharing group.
 - A new LIGHT option of the START DB2 command lets you restart a DB2 data sharing member with a minimal storage footprint, and then terminate normally after DB2 frees the retained locks that it can.
 - You can let changes in structure size persist when you rebuild or reallocate a structure.
 - A new function in z/OS Version 1 Release 2 supports SCA and lock structure duplexing. As a result, a more robust failure recovery is possible in some data sharing environments.
- Additional data sharing enhancements include:
 - Notification of incomplete units of recovery
 - Use of a new OS/390 and z/OS function to improve failure recovery of group buffer pools
- An additional enhancement for e-business provides improved performance with preformatting for INSERT operations.

Easier development and integration of e-business applications

Version 7 provides the following enhancements, which let you more easily develop and integrate applications that access data from various DB2 operating systems and distributed environments:

- DB2 XML Extender for OS/390 and z/OS, a new member of the DB2 Extender family, lets you store, retrieve, and search XML documents in a DB2 database.
- Improved support for UNION and UNION ALL operators in a view definition, a nested table expression, or a subquery predicate, improves DB2 family compatibility and is consistent with SQL99 standards.
- More flexibility with SQL gives you greater compatibility with DB2 on other operating systems:
 - Scrollable cursors let you move forward, backward, or randomly through a result table or a result set. You can use scrollable cursors in any DB2 applications that do not use DB2 private protocol access.
 - A search condition in the WHERE clause can include a subquery in which the base object of both the subquery and the searched UPDATE or DELETE statement are the same.
 - A new SQL clause, FETCH FIRST *n* ROWS, improves performance of applications in a distributed environment.
 - Fast implicit close in which the DB2 server, during a distributed query, automatically closes the cursor when the application attempts to fetch beyond the last row.
 - Support for options USER and USING in a new authorization clause for CONNECT statements lets you easily port applications that are developed on the workstation to DB2 for OS/390 and z/OS. These options also let applications that run under WebSphere to reuse DB2 connections for different users and to enable DB2 for OS/390 and z/OS to check passwords.
 - For positioned updates, you can specify the FOR UPDATE clause of the cursor SELECT statement without a list of columns. As a result, all updatable columns of the table or view that is identified in the first FROM clause of the fullselect are included.
 - A new option of the SELECT statement, ORDER BY *expression*, lets you specify operators as the sort key for the result table of the SELECT statement.
 - New datetime ISO functions return the day of the week with Monday as day 1 and every week with seven days.
- Enhancements to Open Database Connectivity (ODBC) provide partial ODBC 3.0 support, including many new application programming interfaces (APIs), which increase application portability and alignment with industry standards.
- Enhancements to the LOAD utility let you load the output of an SQL SELECT statement directly into a table.
- A new component called Precompiler Services lets compiler writers modify their compilers to invoke Precompiler Services and produce an *SQL statement coprocessor*. An SQL statement coprocessor performs the same functions as the DB2 precompiler, but it performs those functions at compile time. If your compiler has an SQL statement coprocessor, you can eliminate the precompile step in your batch program preparation jobs for C, COBOL, and PL/I programs.
- Support for Unicode-encoded data lets you easily store multilingual data within the same table or on the same DB2 subsystem. The Unicode encoding scheme represents the code points of many different geographies and languages.

#

Improved connectivity

Version 7 offers improved connectivity:

- Support for COMMIT and ROLLBACK in stored procedures lets you commit or roll back an entire unit of work, including uncommitted changes that are made from the calling application before the stored procedure call is made.
- Support for Windows Kerberos security lets you more easily manage workstation clients who seek access to data and services from heterogeneous environments.
- Global transaction support for distributed applications lets independent DB2 agents participate in a global transaction that is coordinated by an XA-compliant transaction manager on a workstation or a gateway server (Microsoft Transaction Server or Encina, for example).
- Support for a DB2 Connect Version 7 enhancement lets remote workstation clients quickly determine the amount of time that DB2 takes to process a request (the server elapsed time).
- Additional enhancements include:
 - Support for connection pooling and transaction pooling for IBM DB2 Connect
 - Support for DB2 Call Level Interface (DB2 CLI) bookmarks on DB2 UDB for UNIX, Windows, OS/2

Features of DB2 for OS/390 and z/OS

Version 7 of DB2 UDB Server for OS/390 and z/OS offers several features that help you integrate, analyze, summarize, and share data across your enterprise:

- #
- #
- #
- DB2 Warehouse Manager feature. The DB2 Warehouse Manager feature brings together the tools to build, manage, govern, and access DB2 for OS/390 and z/OS-based data warehouses. The DB2 Warehouse Manager feature uses proven technologies with new enhancements that are not available in previous releases, including:
 - DB2 Warehouse Center, which includes:
 - DB2 Universal Database Version 7 Release 1 Enterprise Edition
 - Warehouse agents for UNIX, Windows, and OS/390
 - Information Catalog
 - QMF Version 7
 - QMF High Performance Option
 - QMF for Windows
- DB2 Management Clients Package. The elements of the DB2 Management Clients Package are:
 - DB2 Control Center
 - DB2 Stored Procedure Builder
 - DB2 Installer
 - DB2 Visual Explain
 - DB2 Estimator
- Net Search Extender for in-memory text search for e-business applications
- Net.Data for secure Web applications

Migration considerations

- #
- #
- #
- #
- Migration with full fallback protection is available when you have either DB2 for OS/390 Version 5 or Version 6 installed. You should ensure that you are fully operational on DB2 for OS/390 Version 5, or later, before migrating to DB2 for OS/390 and z/OS Version 7.

To learn about all of the migration considerations from Version 5 to Version 7, read the *DB2 Release Planning Guide* for Version 6 and Version 7; to learn about content information, also read appendixes A through F in both books.

Chapter 2. Introduction to DB2 ODBC

DB2 Open Database Connectivity (ODBC) is IBM's callable SQL interface by the DB2 family of products. It is a 'C' and 'C++' application programming interface for relational database access, and it uses function calls to pass dynamic SQL statements as function arguments. It is an alternative to embedded dynamic SQL, but unlike embedded SQL, it does not require a precompiler.

DB2 ODBC is based on the Microsoft® 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 already familiar with either of these *data source* interfaces. In addition, some DB2 specific extensions were added to help the DB2 application programmer specifically exploit DB2 features.

DB2 ODBC background information

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

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

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

The ODBC specification also includes an operating environment where data source specific ODBC drivers are dynamically loaded at run time by a driver manager based on the data source name provided on the connect request. The application is linked directly to a single driver manager library rather than to each DBMS's library. The driver manager mediates the application's function calls at run time and ensures they are directed to the appropriate DBMS specific ODBC driver.

The ODBC driver manager only knows about the ODBC-specific functions, that is, those functions supported by the DBMS for which no API is specified. Therefore, DBMS-specific functions cannot be directly accessed in an ODBC environment. However, DBMS-specific dynamic SQL statements are indirectly supported using a mechanism called the vendor escape clause. See "Using vendor escape clauses" on page 448 for detailed information.

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

Differences between DB2 ODBC and ODBC version 2.0

DB2 ODBC is derived from the ISO Call Level Interface Draft International Standard (ISO CLI DIS) and ODBC Version 2.0.

If you port existing ODBC applications to DB2 for OS/390 and 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 for OS/390 and z/OS and that the syntax and semantics are identical. If there are any differences, you must code to the APIs documented in this publication.

On the DB2 for OS/390 and z/OS platform, no ODBC driver manager exists. Consequently, DB2 ODBC support is implemented as a CLI/ODBC driver/driver manager that is loaded at run time into the application address space. See “DB2 ODBC runtime environment” on page 41 for details about the DB2 ODBC runtime environment.

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

The use of DB2 ODBC in this publication refers to DB2 for OS/390 and z/OS support of DB2 ODBC unless otherwise noted.

General information about DB2 for OS/390 and z/OS is available from the DB2 for OS/390 and z/OS World Wide Web page:

<http://www.software.ibm.com/data/db2/os390/>

ODBC features supported

DB2 ODBC support should be viewed as consisting of most of ODBC Version 2.0 as well as IBM extensions. Where differences exist, applications should be written to the specifications defined in this publication.

DB2 ODBC includes support of the following ODBC functions:

- All ODBC level 1 functions
- All ODBC level 2 functions with the following four exceptions:
 - SQLBrowseConnect()
 - SQLDrivers()
 - SQLSetPos()
 - SQLSetScrollOptions()
- Some X/Open CLI functions
- Some DB2 specific functions

For a complete list of supported functions, see “Function summary” on page 68.

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

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 null termination of output strings.
- Support of large objects (LOBs) and LOB locators

DB2 ODBC does not support the following functions (a deviation from the Microsoft ODBC Version 2.0 Specification):

- Asynchronous SQL
- Scrollable cursor support
- Interactive data source connection specified using `SQLBrowseConnect()` and `SQLDriverConnect()`.

For more information on the relationship between DB2 ODBC and ODBC, see Appendix A, “DB2 ODBC and ODBC”, on page 467.

Differences between DB2 ODBC and embedded SQL

An application that uses an embedded SQL interface requires a precompiler to convert the SQL statements into code, which is then compiled, bound to the data source, and executed. In contrast, a DB2 ODBC application does not have to be precompiled or bound, but instead uses a standard set of functions to execute SQL statements and related services at run time.

This difference is important because, traditionally, precompilers have been specific to each database product, which effectively ties your applications to that product. DB2 ODBC enables you to write portable applications that are independent of any particular database product. The DB2 ODBC driver contains a fixed set of precompiler options. Therefore, the settings of DB2 for OS/390 and z/OS precompiler options have no affect on ODBC behavior.

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, there are differences.

(There are also differences between ODBC SQLSTATEs and the X/Open defined SQLSTATEs). Refer to Table 173 on page 475 for a cross reference of all DB2 ODBC SQLSTATEs.

Despite these differences, there is an important common concept between embedded SQL and DB2 ODBC:

DB2 ODBC can execute any SQL statement that can be prepared dynamically in embedded SQL.

Table 1 lists each DB2 for OS/390 and z/OS SQL statement, and indicates whether or not it can be executed using DB2 ODBC.

Each DBMS might have additional statements that can be dynamically prepared, in which case DB2 ODBC passes them to the DBMS. There is one exception: COMMIT and ROLLBACK can be dynamically prepared by some DBMSs but are not passed. The SQLEndTran() function should be used instead to specify either COMMIT or ROLLBACK.

Table 1. SQL statements

SQL statement	Dynamic ^a	DB2 ODBC ^c
ALTER TABLE	X	X
ALTER DATABASE	X	X
ALTER INDEX	X	X
ALTER STOGROUP	X	X
ALTER TABLESPACE	X	X
BEGIN DECLARE SECTION ^b		
CALL		X ^d
CLOSE		SQLFreeHandle()
COMMENT ON	X	X
COMMIT	X	SQLEndTran()
CONNECT (type 1)		SQLConnect(), SQLDriverConnect()
CONNECT (type 2)		SQLConnect(), SQLDriverConnect()
CREATE { ALIAS, DATABASE, INDEX, STOGROUP, SYNONYM, TABLE, TABLESPACE, VIEW, DISTINCT TYPE }	X	X
DECLARE CURSOR ^b		SQLAllocHandle()
DECLARE STATEMENT		
DECLARE TABLE		
DELETE	X	X
DESCRIBE		SQLDescribeCol(), SQLColAttribute()
DROP	X	X
END DECLARE SECTION ^b		
EXECUTE		SQLExecute()
EXECUTE IMMEDIATE		SQLExecDirect()
EXPLAIN	X	X
FETCH		SQLFetch()
FREE LOCATOR ^d		X

Table 1. SQL statements (continued)

SQL statement	Dynamic ^a	DB2 ODBC ^c
GRANT	X	X
HOLD LOCATOR ^d		X
INCLUDE ^b		
INSERT	X	X
LABEL ON	X	X
LOCK TABLE	X	X
OPEN		SQLExecute(), SQLExecDirect()
PREPARE		SQLPrepare()
RELEASE		
REVOKE	X	X
ROLLBACK	X	SQLEndTran()
select-statement	X	X
SELECT INTO		
SET CONNECTION		SQLSetConnection()
SET host_variable		
SET CURRENT DEGREE	X	X
SET CURRENT PACKAGESET		
SET CURRENT PATH	X	X
SET CURRENT SQLID	X	X
UPDATE	X	X
WHENEVER ^b		

Note:

^a All statements in this list can be coded as static SQL, but only those marked with X can be coded as dynamic SQL.

^b This statement is not executable.

^c An X indicates that this statement can be executed using either SQLExecDirect() or SQLPrepare() and SQLExecute(). If there is an equivalent DB2 ODBC function, the function name is listed.

^d Although this statement is not dynamic, DB2 ODBC allows the statement to be specified when calling either SQLExecDirect() or SQLPrepare() and SQLExecute().

Advantages of using DB2 ODBC

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

- Ideally suits the client-server environment in which the target data source is unknown when the application is built. It provides a consistent interface for executing SQL statements, regardless of which database server the application connects to.
- Lets you write portable applications that are independent of any particular database product. DB2 ODBC applications do not have to be recompiled or rebound to access different DB2 or DRDA data sources. Instead they connect to the appropriate data source at run time.

- Reduces the amount of management required for an application while in general use. Individual DB2 ODBC applications do not need to be bound to each data source. Bind files provided with DB2 ODBC need to be bound only once for all DB2 ODBC applications.
- Lets applications connect to multiple data sources from the same application.
- Allocates and controls data structures, and provides a handle for the application to refer to them. Applications do not have to control complex global data areas such as the SQLDA and SQLCA.
- Provides enhanced parameter input and fetching capability. You can specify arrays of data on input to retrieve multiple rows of a result set directly into an array. You can execute statements that generate multiple result sets.
- Lets you retrieve multiple rows and result sets generated from a call to a stored procedure.
- Provides a consistent interface to query catalog information that is contained in various DBMS catalog tables. The result sets that are returned are consistent across DBMSs. Application programmers can avoid writing version-specific and server-specific catalog queries.
- Provides extended data conversion which requires less application code when converting information between various SQL and C data types.
- Aligns with the emerging ISO CLI standard in addition to using the accepted industry specifications of ODBC and X/Open CLI.
- Allows application developers to apply their knowledge of industry standards directly to DB2 ODBC. The interface is intuitive for programmers who are familiar with function libraries but know little about product specific methods of embedding SQL statements into a host language.

Deciding which interface to use

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

Static and dynamic SQL

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

- Performance

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

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

- Encapsulation and security

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

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

Use both interfaces

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

Write a mixed application

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

Other information sources

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

When writing DB2 ODBC applications, you also might need to reference information for the database servers that are being accessed, in order to understand any connectivity issues, environment issues, SQL language support issues, and other server-specific information. For DB2 for OS/390 and z/OS versions, see *DB2 SQL Reference* and *DB2 Application Programming and SQL Guide*. If you are writing applications that access other DB2 server products, see *IBM SQL Reference* for information that is common to all products, including any differences.

Chapter 3. Writing a DB2 ODBC application

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

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

Tasks described in this section are basic tasks that apply to all applications. More advanced tasks, such as using array insert, are described in Chapter 6, “Using advanced features”, on page 389.

The functions are used in examples to illustrate their use in DB2 ODBC applications. See Chapter 5, “Functions”, on page 67 for complete descriptions and usage information for each of the functions.

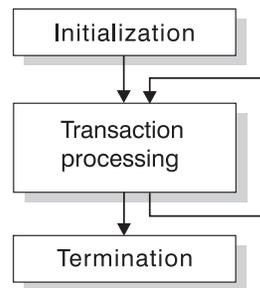


Figure 1. Conceptual view of a DB2 ODBC application

Every DB2 ODBC application contains the three main tasks shown in Figure 1.

Initialization

This task allocates and initializes some resources in preparation for the main *transaction processing* task. See “Initialization and termination” on page 16 for details.

Transaction processing

This is the main task of the application. SQL statements are passed to DB2 ODBC to query and modify the data. See “Transaction processing” on page 20 for details.

Termination

This task frees allocated resources. The resources generally consist of data areas identified by unique handles. See “Initialization and termination” on page 16 for details.

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

Initialization and termination

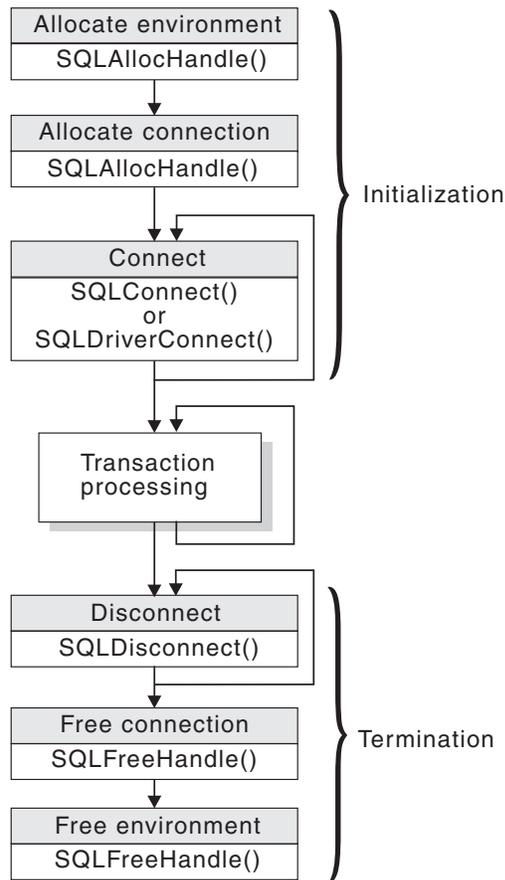


Figure 2. Conceptual view of initialization and termination tasks

Figure 2 shows the function call sequences for both the initialization and termination tasks. The transaction processing task in the middle of the diagram is shown in Figure 3 on page 21.

Handles

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

There are three types of handles:

Environment handle

The environment handle refers to the data object that contains information regarding the global state of the application, such as attributes and connections. 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 options, general status information, transaction status, and diagnostic information. Each connection handle is allocated by calling SQLAllocHandle() (with *HandleType* set to SQL_HANDLE_DBC) and freed by calling SQLFreeHandle() (with *HandleType* set to SQL_HANDLE_DBC).

An application can be connected to several database servers at the same time. An application requires a connection handle for each concurrent connection to a database server. For information on multiple connections, see “Connecting to one or more data sources” on page 18.

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

Statement handles

Statement handles are discussed in the next section, “Transaction processing” on page 20.

ODBC connection model

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

DB2 ODBC restrictions on the ODBC connection model

If the application is not using the MULTICONTTEXT=1 initialization file setting, there are restrictions on DB2 ODBC’s support of the ODBC connection model. To obtain simulated support of the ODBC connection model, the application must specify a CONNECT type value of 1 (either by using the initialization file or the SQLSetConnectAttr() API. See “Initialization keywords” on page 55 and “Specifying the connect type” on page 18.)

The application can then logically connect to any number of data sources. However, the DB2 ODBC driver maintains only a single physical connection, that of the last data source to which the application successfully connected or at which the last SQL statement was executed.

As a result, the application is affected as follows:

- When connected to one or more data sources so that the application has allocated some number of connect handles, any attempt to connect to a new data source commits the work on the current data source and terminates that connection. Therefore, the application cannot have cursors concurrently open at two data sources (including cursors WITH HOLD).
- If the application is currently connected to X and has performed work at X that has not yet been committed or rolled back, then any execution of an API to perform work on a valid statement handle Y results in committing the transaction at X and reestablishing the connection to Y.

With multiple context support, DB2 ODBC can fully support the ODBC connection model. See “DB2 ODBC support of multiple contexts” on page 424.

Connect type 1 and type 2

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

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

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

ODBC does not support multiple connections participating in a distributed transaction.

Specifying the connect type

Important: The connect type must be established prior to issuing `SQLConnect()`.

You can establish the connect type using either of the following methods:

- Specify the `CONNECTTYPE` keyword in the common section of the initialization file with a value of 1 (CONNECT (type 1)) or 2 (CONNECT (type 2)). The initialization file is described in “DB2 ODBC initialization file” on page 52.
- Invoke `SQLSetConnectAttr()`. Specify `fOption = SQL_CONNECTTYPE` with a value of `SQL_CONCURRENT_TRANS` (CONNECT (type 1)) or a value of `SQL_COORDINATED_TRANS` (CONNECT (type 2)).

Connecting to one or more data sources

DB2 ODBC supports connections to remote data sources through DRDA.

If the application is executing with CONNECT (type 1) and `MULTICONTEXT=0`, then DB2 ODBC allows an application to *logically* connect to multiple data sources; however, all transactions other than the transaction associated with the current connection, must be complete (committed or rolled back). If the application is executing with 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, an application calls `SQLAllocHandle()` (with `HandleType` set to `SQL_HANDLE_DBC`) once for each connection. The subsequent connection handle is used with `SQLConnect()` to request a data source connection and with `SQLAllocHandle()` (with `HandleType` set to `SQL_HANDLE_STMT`) to allocate statement handles for use within that connection. There is also an extended connect function, `SQLDriverConnect()`, which allows for additional connect options.

Unlike the distributed unit of work connections described in “Distributed unit of work (coordinated distributed transactions)” on page 391, there is no coordination between the statements that are executed on different connections.

Initialization and connection example

```
/* ... */
/*****
**   - demonstrate basic connection to two datasources.
**   - 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, etc. */
/***** 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 USERID and PASSWORD**
** are always NULL. **
*****/

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

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

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

```

Transaction processing

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

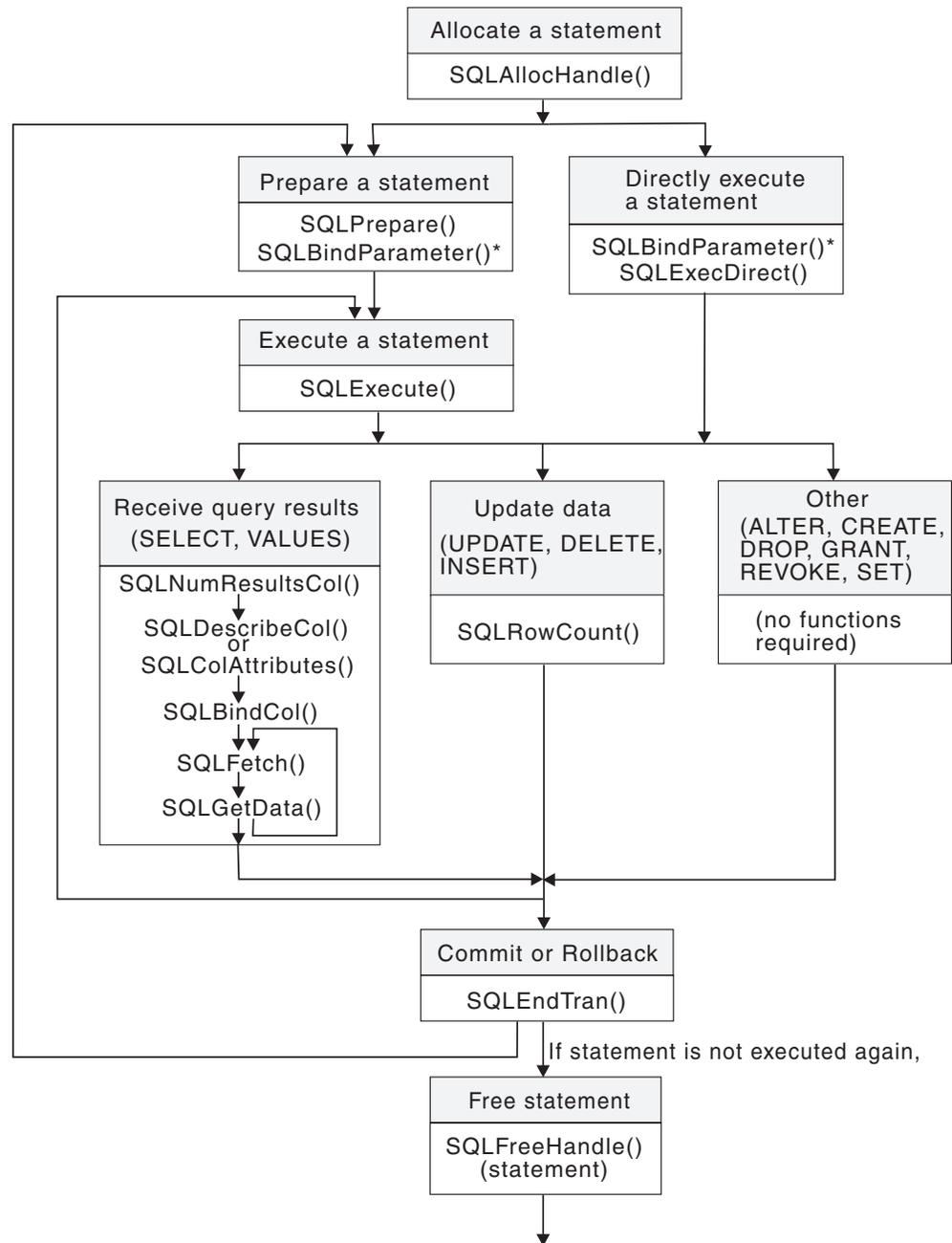


Figure 3. Transaction processing

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

- Allocating statement handles
- Preparation and execution of SQL statements
- Processing result
- Commit or rollback
- Optionally, freeing statement handles if the statement is unlikely to be executed again.

Generally, you can use the `SQLSetParam()` function in place of the `SQLBindParameter()` function. `SQLSetParam()` is slightly simpler.

Recommendation: Use the `SQLBindParameter()` function which is more current.

Allocating statement handles

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

A statement handle must be allocated before a statement can be executed. By default, the maximum number of statement handles that can be allocated at any one time is limited by the application heap size. The maximum number of statement handles that can actually be used, however, is defined by DB2 ODBC. Table 2 lists the number of statement handles allowed for each isolation level. If an application exceeds these limits, `SQLSTATE S1014` is returned on the call to `SQLPrepare()` or `SQLExecuteDirect()`.

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

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

Preparation and execution

After a statement handle is allocated, there are two methods of specifying and executing SQL statements:

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

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

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

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

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

DB2 for OS/390 and z/OS and DB2 UDB provide *dynamic statement caching* at the database server. In DB2 ODBC terms this means that for a given statement handle, once a statement is prepared, it does not need to be prepared again (even after commits or rollbacks), as long as the statement handle is not freed. Applications that repeatedly execute the same SQL statement across multiple transactions, can save a significant amount of processing time and network traffic by:

1. Associating each such statement with its own statement handle, and
2. Preparing these statements once at the beginning of the application, then
3. Executing the statements as many times as is needed throughout the application.

Binding parameters in SQL statements

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

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

When an application variable is associated with a parameter marker it is *bound* to the parameter marker. The application must bind an application variable to each parameter marker in the SQL statement before it executes that statement. Binding is carried out by calling the `SQLBindParameter()` function with a number of arguments to indicate, the numerical position of the parameter, the SQL type of the parameter, the data type of the variable, a pointer to the application variable, and length of the variable. For UCS-2 application variables, applications must bind parameter markers to the `SQL_C_WCHAR` data type.

The bound application variable and its associated length are called *deferred* input arguments since only the pointers are passed when the parameter is bound; no data is read from the variable until the statement is executed. Deferred arguments allow the application to modify the contents of the bound parameter variables, and repeat the execution of the statement with the new values.

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

It is possible to bind the parameters to a variable of a different type from that required by the SQL statement. The application must indicate the C data type of the source, and the SQL type of the parameter marker, and DB2 ODBC converts the contents of the variable to match the SQL data type specified. For example, the

SQL statement might require an integer value, but your application has a string representation of an integer. The string can be bound to the parameter, and DB2 ODBC converts the string to the corresponding integer value when you execute the statement.

`SQLDescribeParam()` can be used to determine the data type of a parameter marker. If the application indicates an incorrect type for the parameter marker, either an extra conversion by the DBMS, or an error can occur. See “Data types and data conversion” on page 30 for more information about data conversion.

A parameter marker that is part of a predicate on a query and is associated with a distinct type must be cast to the built-in type in the predicate portion of the statement. Otherwise, an error occurs. For an example of casting distinct types, see “Casting distinct types” on page 455.

For information on more advanced methods of binding application storage to parameter markers, see “Using arrays to input parameter values” on page 403 and “Sending/retrieving long data in pieces” on page 401.

Processing results

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

Processing query (SELECT, VALUES) statements

If the statement is a query statement, the following steps are generally needed in order to retrieve each row of the result set:

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

Each of the above steps requires some diagnostic checks. Chapter 6, “Using advanced features”, on page 389 discusses advanced techniques of using `SQLExtendedFetch()` to fetch multiple rows at a time.

Step 1

Analyze the executed or prepared statement. If the SQL statement was generated by the application, then this step might not be necessary since the application might know the structure of the result set and the data types of each column. If the application does know the structure of the entire result set, and if there are a very large number of columns to retrieve, then the application might wish to supply DB2 ODBC with the descriptor information. This can reduce network traffic since DB2 ODBC does not have to retrieve the information from the server.

On the other hand, if the SQL statement was generated at runtime (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 `SQLColAttribute()`) after preparing or after executing the statement.

Step 2

The application retrieves column data directly into an application variable on

the next call to `SQLFetch()`. For each column to be retrieved, the application calls `SQLBindCol()` to bind an application variable to a column in the result set. The application can use the information obtained from Step 1 to determine the C data type of the application variable and to allocate the maximum storage the column value could occupy. Similar to variables bound to parameter markers using `SQLBindParameter()` and `SQLSetParam()`, columns are bound to deferred arguments. This time the variables are deferred output arguments, as data is written to these storage locations when `SQLFetch()` is called.

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

Step 3

Call `SQLFetch()` to fetch the first or next row of the result set. If any columns are bound, the application variable is updated. There is also a method that allows the application to fetch multiple rows of the result set into an array, see “Retrieving a result set into an array” on page 406 for more information.

If data conversion was indicated by the data types specified on the call to `SQLBindCol()`, the conversion occurs when `SQLFetch()` is called. See “Data types and data conversion” on page 30 for an explanation.

Step 4 (optional)

Call `SQLGetData()` to retrieve any unbound columns. All columns can be retrieved this way, provided they were not bound. `SQLGetData()` can also be called repeatedly to retrieve large columns in smaller pieces, which cannot be done with bound columns.

Data conversion can also be indicated here, as in `SQLBindCol()`, by specifying the desired target C data type of the application variable. See “Data types and data conversion” on page 30 for more information.

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

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

For information on more advanced methods for binding application storage to result set columns, see “Retrieving a result set into an array” on page 406 and “Sending/retrieving long data in pieces” on page 401.

Processing UPDATE, DELETE and INSERT statements

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

If the SQL statement is a positioned UPDATE or DELETE, it is necessary to use a *cursor*. A cursor is a moveable pointer to a row in the result table of an active query statement. (This query statement must contain the FOR UPDATE OF clause to ensure that the query is not opened as read-only.) In embedded SQL, cursor names are used to retrieve, update or delete rows. In DB2 ODBC, a cursor name is needed only for positioned UPDATE or DELETE SQL statements as they reference the cursor by name.

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

Processing other statements

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

Commit or rollback

A *transaction* is a recoverable unit of work, or a group of SQL statements that can be treated as one atomic operation. This means that all the operations within the group are guaranteed to be completed (committed) or undone (rolled back), as if they were a single operation. A transaction can also be referred to as a unit of work or a logical unit of work. When the transaction spans multiple connections, it is referred to as a distributed unit of work.

DB2 ODBC supports two commit modes: *auto-commit* and *manual-commit*.

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

In manual-commit mode, transactions are started implicitly with the first access to the data source using `SQLPrepare()`, `SQLExecDirect()`, `SQLGetTypeInfo()`, or any function that returns a result set, such as those described in “Querying system catalog information” on page 397. 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 auto-commit (except when participating in a coordinated transaction, see “Distributed unit of work (coordinated distributed transactions)” on page 391). An application can switch between manual-commit and auto-commit modes by calling `SQLSetConnectAttr()`. Typically, a query-only application might wish to stay in auto-commit mode. Applications that need to perform updates to the data source should turn off auto-commit as soon as the data source connection is established.

When multiple connections exist, each connection has its own transaction (unless `CONNECT` (type 2) is specified). Special care must be taken to call `SQLEndTran()`

with the correct connection handle to ensure that only the intended connection and related transaction is affected. Unlike distributed unit of work connections (described in “Distributed unit of work (coordinated distributed transactions)” on page 391), there is no coordination between the transactions on each connection.

When to call `SQLEndTran()`

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

In manual-commit mode, `SQLEndTran()` must be called before calling `SQLDisconnect()`. If distributed unit of work is involved, additional rules can apply. See “Distributed unit of work (coordinated distributed transactions)” on page 391 for details.

Recommendation: An application that performs updates should not wait until the disconnect before committing or rolling back a transaction.

The other extreme is to operate in auto-commit mode, which is also not recommended as this adds extra processing. The application can modify the auto-commit mode by invoking the `SQLSetConnectAttr()` function. See “Environment, connection, and statement options” on page 389 and the `SQLSetConnectAttr()` function for information about switching between auto-commit and manual-commit.

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

- If using `CONNECT` (type 1) with `MULTICONTEXT=0`, only the current connection can have an outstanding transaction. If using `CONNECT` (type 2), all connections participate in a single transaction.
- If using `MULTICONTEXT=1`, each connection can have an outstanding transaction.
- Various resources can be held while you have an outstanding transaction. Ending the transaction releases the resources for use by other users.
- When a transaction is successfully committed or rolled back, it is fully recoverable from the system logs. Open transactions are not recoverable.

Effects of calling `SQLEndTran()`

When a transaction ends:

- All locks on DBMS objects are released, except those that are associated with a held cursor.
- Prepared statements are preserved from one transaction to the next if the data source supports statement caching (DB2 for OS/390 and z/OS Version 5 does). After a statement is prepared on a specific statement handle, it does not need to be prepared again even after a commit or rollback, provided the statement continues to be associated with the same statement handle.
- Cursor names, bound parameters, and column bindings are maintained from one transaction to the next.
- By default, cursors are preserved after a commit (but not a rollback). All cursors are defined using the `WITH HOLD` clause (except when connected to DB2 Server for VSE & VM, which does not support the `WITH HOLD` clause). For information about changing the default behavior, see “`SQLSetStmtOption` - Set statement option” on page 367.

For more information and an example see “`SQLTransact` - Transaction management” on page 386.

Freeing statement handles

Call `SQLFreeHandle()` (with *HandleType* set to `SQL_HANDLE_STMT`) to end processing for a particular statement handle. This function can be used to do one or more of the following:

- Unbind all columns of the result set
- Unbind all parameter markers
- Close any cursors and discard any pending results
- Drop the statement handle, and release all associated resources

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

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

Diagnostics

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

- 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 three pieces of information:

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

For the detailed function information and example usage, see “SQLError - Retrieve error information” on page 155.

For diagnostic information about DB2 ODBC traces and debugging, see Chapter 7, “Problem diagnosis”, on page 457.

Function return codes

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 <code>SQLGetDiagRec()</code> to receive the SQLSTATE and any other informational messages or warnings. The SQLSTATE class is '01'. See Table 173 on page 475.

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 <code>SQLGetDiagRec()</code> .
SQL_NEED_DATA	The application tried to execute an SQL statement but DB2 ODBC lacks parameter data that the application had indicated would be passed at execute time. For more information, see “Sending/retrieving long data in pieces” on page 401.
SQL_ERROR	The function failed. Call <code>SQLGetDiagRec()</code> to receive the <code>SQLSTATE</code> and any other error information.
SQL_INVALID_HANDLE	The function failed due to an invalid input handle (environment, connection or statement handle). This is a programming error. No further information is available.

SQLSTATEs

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

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

- '01', is a warning.
- 'S1', is generated by the DB2 ODBC driver.

Note: X/Open has reserved class 'HY' for ODBC/CLI implementations, which is currently equivalent to the 'S1' class. This might be a consideration if you intend to follow the X/Open and/or ISO CLI standard in the future.

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

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

Follow these guidelines for using *SQLSTATEs* within your application:

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

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

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

See Table 173 on page 475 for a listing and description of the SQLSTATEs explicitly returned by DB2 ODBC.

SQLCA

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

The `SQLGetSQLCA()` function is used to retrieve this structure. See “SQLGetSQLCA - Get SQLCA data structure” on page 263 for more information.

Data types and data conversion

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

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

C and SQL data types

Table 4 on page 31 lists each of the SQL data types, with its corresponding symbolic name, and the default C symbolic name. These data types represent the combination of the ODBC V2.0 minimum, core, and extended data types. The ODBC extended data type `SQL_BIGINT` is not supported. In addition, DB2 ODBC supports `SQL_GRAPHIC`, `SQL_VARGRAPHIC` and `SQL_LONGVARGRAPHIC`.

SQL data type

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

Symbolic SQL data type

This column contains SQL symbolic names that are defined (in `sqlcli1.h`) as an integer value. These values are used by various functions to identify the SQL data types listed in the first column. See “Example” on page 139 for an example using these values.

Default C symbolic data type

This column contains C symbolic names, also defined as integer values. These values are used in various function arguments to identify the C data type as shown in Table 5 on page 32. The symbolic names are used by various functions, (such as `SQLBindParameter()`, `SQLGetData()`, `SQLBindCol()`) to indicate the C data types of the application variables. Instead of explicitly identifying the C data type when calling these functions, `SQL_C_DEFAULT` can be specified instead, and DB2 ODBC assumes a default C data type based on the SQL data type of the parameter or column as shown by this table. For example, the default C data type of `SQL_DECIMAL` is `SQL_C_CHAR`.

Table 4. SQL symbolic and default data types

SQL data type	Symbolic SQL data type	Default symbolic C data type
BLOB	SQL_BLOB	SQL_C_BINARY
BLOB LOCATOR ^a	SQL_BLOB_LOCATOR	SQL_C_BLOB_LOCATOR
CHAR	SQL_CHAR	SQL_C_CHAR
CHAR FOR BIT DATA	SQL_BINARY	SQL_C_BINARY
CLOB	SQL_CLOB	SQL_C_CHAR
CLOB LOCATOR ^a	SQL_CLOB_LOCATOR	SQL_C_CLOB_LOCATOR
DATE	SQL_TYPE_DATE ^e	SQL_C_TYPE_DATE ^e
DBCLOB	SQL_DBCLOB	SQL_C_DBCHAR
DBCLOB LOCATOR ^a	SQL_DBCLOB_LOCATOR	SQL_C_DBCLOB_LOCATOR
DECIMAL	SQL_DECIMAL	SQL_C_CHAR
DOUBLE	SQL_DOUBLE	SQL_C_DOUBLE
FLOAT	SQL_FLOAT	SQL_C_DOUBLE
GRAPHIC	SQL_GRAPHIC	SQL_C_DBCHAR
INTEGER	SQL_INTEGER	SQL_C_LONG
LONG VARCHAR ^b	SQL_LONGVARCHAR	SQL_C_CHAR
LONG VARCHAR FOR BIT DATA ^b	SQL_LONGVARBINARY	SQL_C_BINARY
LONG VARGRAPHIC ^b	SQL_LONGVARGRAPHIC	SQL_C_DBCHAR
NUMERIC ^c	SQL_NUMERIC ^c	SQL_C_CHAR
REAL ^d	SQL_REAL	SQL_C_FLOAT
ROWID	SQL_ROWID	SQL_C_CHAR
SMALLINT	SQL_SMALLINT	SQL_C_SHORT
TIME	SQL_TYPE_TIME ^e	SQL_C_TYPE_TIME ^e
TIMESTAMP	SQL_TYPE_TIMESTAMP ^e	SQL_C_TYPE_TIMESTAMP ^e
VARCHAR	SQL_VARCHAR	SQL_C_CHAR
VARCHAR FOR BIT DATA ^b	SQL_VARBINARY	SQL_C_BINARY
VARGRAPHIC	SQL_VARGRAPHIC	SQL_C_DBCHAR

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

SQL data type	Symbolic SQL data type	Default symbolic C data type
---------------	------------------------	------------------------------

Note:

- a** LOB locator types are not persistent SQL data types (columns cannot be defined by a locator type; instead, it describes parameter markers, or represents a LOB value). See “Using large objects” on page 411 for more information.
- b** Whenever possible, replace long data types and FOR BIT DATA data types with appropriate LOB types.
- c** NUMERIC is a synonym for DECIMAL on DB2 for OS/390 and z/OS, DB2 for VSE & VM and DB2 UDB.
- d** REAL is not valid for DB2 UDB or DB2 for OS/390 and z/OS.
- e** See “Changes to datetime data types” on page 505 for information about data types used in previous releases.

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

Table 5 shows the generic C type definitions for each symbolic C type.

C symbolic data type

This column contains C symbolic names, defined as integer values. These values are used in various function arguments to identify the C data type shown in the last column. See “Example” on page 90 for an example using these values.

C type

This column contains C defined types, defined in `sqlcli1.h` using a C typedef statement. The values in this column should be used to declare all DB2 ODBC related variables and arguments, in order to make the application more portable. See Table 7 on page 33 for a list of additional symbolic data types used for function arguments.

Base C type

This column is shown for reference only. All variables and arguments should be defined using the symbolic types in the previous column. Some of the values are C structures that are described in Table 6 on page 33.

Table 5. C data types

C symbolic data type	C type	Base C type
SQL_C_CHAR	SQLCHAR	unsigned char
SQL_C_BIT	SQLCHAR	unsigned char or char (Value 1 or 0)
SQL_C_TINYINT	SQLSCHAR	signed char (Range -128 to 127)
SQL_C_SHORT	SQLSMALLINT	short int
SQL_C_LONG	SQLINTEGER	long int
SQL_C_DOUBLE	SQLDOUBLE	double
SQL_C_FLOAT	SQLREAL	float
SQL_C_TYPE_DATE ^a	DATE_STRUCT	see Table 6 on page 33
SQL_C_TYPE_TIME ^a	TIME_STRUCT	see Table 6 on page 33
SQL_C_TYPE_TIMESTAMP ^a	TIMESTAMP_STRUCT	see Table 6 on page 33
SQL_C_CLOB_LOCATOR	SQLINTEGER	long int
SQL_C_BINARY	SQLCHAR	unsigned char
SQL_C_BLOB_LOCATOR	SQLINTEGER	long int

Table 5. C data types (continued)

C symbolic data type	C type	Base C type
SQL_C_DBCHAR	SQLDBCHAR	wchar_t
SQL_C_DBCLOB_LOCATOR	SQLINTEGER	long int
SQL_C_WCHAR	SQLWCHAR	wchar_t

Note: a See “Changes to datetime data types” on page 505 for information about data types used in previous releases.

Table 6. C date, time, and timestamp structures

C type	Generic structure
DATE_STRUCT	<pre>typedef struct DATE_STRUCT { SQLSMALLINT year; SQLUSMALLINT month; SQLUSMALLINT day; } DATE_STRUCT;</pre>
TIME_STRUCT	<pre>typedef struct TIME_STRUCT { SQLUSMALLINT hour; SQLUSMALLINT minute; SQLUSMALLINT second; } TIME_STRUCT;</pre>
TIMESTAMP_STRUCT	<pre>typedef struct TIMESTAMP_STRUCT { SQLUSMALLINT year; SQLUSMALLINT month; SQLUSMALLINT day; SQLUSMALLINT hour; SQLUSMALLINT minute; SQLUSMALLINT second; SQLINTEGER fraction; } TIMESTAMP_STRUCT;</pre>

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

Other C data types

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

Table 7. C data types and base C data types

Defined C type	Base C type	Typical usage
SQLPOINTER	void *	Pointers to storage for data and parameters.
SQLHENV	long int	Handle referencing environment information.
SQLHDBC	long int	Handle referencing data source connection information.
SQLHSTMT	long int	Handle referencing statement information.
SQLUSMALLINT	unsigned short int	Function input argument for unsigned short integer values.
SQLINTEGER	unsigned long int	Function input argument for unsigned long integer values.

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

Defined C type	Base C type	Typical usage
SQLRETURN	short int	Return code from DB2 ODBC functions.
SQLWCHAR	wchar_t	Function input argument for suffix-W API use.

Data conversion

As mentioned previously, DB2 ODBC manages the transfer and any required conversion of data between the application and the DBMS. Before the data transfer actually takes place, the source, target or both data types are indicated when calling `SQLBindParameter()`, `SQLBindCol()` or `SQLGetData()`. These functions use the symbolic type names shown in Table 4 on page 31, to identify the data types involved.

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

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

Table 4 shows only the default data conversions. The functions mentioned in the previous paragraph can be used to convert data to other types, but not all data conversions are supported or make sense. Table 8 on page 35 shows all the conversions supported by DB2 ODBC.

The first column in Table 8 contains the data type of the SQL data type, the remaining columns represent the C data types. If the C data type columns contains:

- D** The conversion is supported and is the default conversion for the SQL data type.
- X** All IBM DBMSs support the conversion,
- Blank** No IBM DBMS supports the conversion.

For example, the table indicates that a `char` (or a C character string as indicated in Table 8) can be converted into a `SQL_C_LONG` (a signed long). In contrast, a `LONGVARCHAR` cannot be converted to a `SQL_C_LONG`.

See Appendix D, "Data conversion", on page 485 for information about required formats and the results of converting between data types.

Limits on precision, and scale, as well as truncation and rounding rules for type conversions follow rules specified in the *IBM SQL Reference* with the following exception; truncation of values to the right of the decimal point for numeric values returns a truncation warning, whereas truncation to the left of the decimal point returns an error. In cases of error, the application should call `SQLGetDiagRec()` to obtain the `SQLSTATE` and additional information on 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.

Table 8. Supported data conversions

SQL data type	SQL_C_CHAR	SQL_C_WCHAR	SQL_C_LONG	SQL_C_SHORT	SQL_C_TINYINT	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_TYPE_DATE	SQL_C_TYPE_TIME	SQL_C_TYPE_TIMESTAMP	SQL_C_BINARY	SQL_C_BIT	SQL_C_DBCHAR	SQL_C_CLOB_LOCATOR	SQL_C_BLOB_LOCATOR	SQL_C_DBCLOB_LOCATOR
SQL_BLOB	X										D				X	
SQL_CHAR	X		X	X	X	X	X	X	X	X	X	X				
SQL_CLOB	D										X			X		
SQL_TYPE_DATE	X							D		X						
SQL_DBCLOB		D									X		D			X
SQL_DECIMAL	D		X	X	X	X	X					X				
SQL_DOUBLE	X		X	X	X	X	D					X				
SQL_FLOAT	X		X	X	X	X	D					X				
SQL_GRAPHIC	X	D											D			
SQL_INTEGER	X		D	X	X	X	X					X				
SQL_LONGVARCHAR	D							X		X	X					
SQL_LONGVARGRAPHIC	X	D									X		D			
SQL_NUMERIC	D		X	X	X	X	X					X				
SQL_REAL	X		X	X	X	D	X					X				
SQL_ROWID	X															
SQL_SMALLINT	X		X	D	X	X	X					X				
SQL_TYPE_TIME	X								D	X						
SQL_TYPE_TIMESTAMP	X							X	X	D						
SQL_VARCHAR	D		X	X	X	X	X	X	X	X	X	X				
SQL_VARGRAPHIC	X												D			

Note:

- Data is not converted to LOB locator types; locators represent a data value.
- REAL is not supported by DB2 UDB.
- NUMERIC is a synonym for DECIMAL on DB2 for OS/390 and z/OS, DB2 for VSE & VM, and DB2 UDB.
- The application must bind data to the SQL_C_WCHAR data type for UCS-2 data. See “Using Unicode functions” on page 429 for more information.

Working with string arguments

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

Length of string arguments

Input string arguments have an associated length argument. This argument indicates to DB2 ODBC, either the exact length of the argument (not including the null terminator), the special value SQL_NTS to indicate a null-terminated string, or SQL_NULL_DATA to pass a NULL value. If the length is set to SQL_NTS, DB2 ODBC determines the length of the string by locating the null terminator. For a UCS-2 string, SQL_NTS indicates the string length, including a two-byte null terminator.

Output string arguments have two associated length arguments, an input length argument to specify the length of the allocated output buffer, and an output length argument to return the actual length of the string returned by DB2 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. It is highly recommended that a valid output length argument always be used.

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;
```

Null-termination of strings

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

It is also possible to use `SQLSetEnvAttr()` and set an environment attribute to disable null termination of variable length output (character string) data. In this case, the application allocates a buffer exactly as long as the longest string it expects. The application must provide a valid pointer to storage for the output length argument so that DB2 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 null terminator.

String truncation

If an output string does not fit into a buffer, DB2 ODBC truncates the string to the size of the buffer, and writes the null terminator. If truncation occurs, the function returns `SQL_SUCCESS_WITH_INFO` and an `SQLSTATE` of `01004` indicating truncation. The application can then compare the buffer length to the output length to determine which string was truncated.

For example, if `SQLFetch()` returns `SQL_SUCCESS_WITH_INFO`, and an `SQLSTATE` of `01004`, 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, an IBM relational database (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 an SQLSTATE of 22001. On output, DB2 truncates the data and issues SQL_SUCCESS_WITH_INFO and an SQLSTATE of 01004.

Interpretation of strings

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

Querying environment and data source information

There are many situations when an application requires information about the characteristics and capabilities of the current DB2 ODBC driver or the data source that it is connected to.

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

These functions are also useful to generic applications that are written to adapt and take advantage of facilities that might be available from some, but not all database servers. The following DB2 ODBC functions provide data source specific information:

- “SQLDataSources - Get list of data sources” on page 134
- “SQLGetFunctions - Get functions” on page 228
- “SQLGetInfo - Get general information” on page 234
- “SQLGetTypeInfo - Get data type information” on page 278

Querying environment information example

```
/******  
/* Querying environment and data source information */  
/******  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
#include <sqlcli1.h>  
  
void main()  
{  
    SQLHENV        hEnv;           /* Environment handle          */  
    SQLHDBC        hDbc;           /* Connection handle          */  
    SQLRETURN      rc;             /* Return code for API calls  */  
    SQLHSTMT       hStmt;          /* Statement handle           */  
    SQLCHAR        dsname[30];     /* Data source name            */  
    SQLCHAR        dsdescr[255];   /* Data source description     */  
    SQLSMALLINT    dslen;          /* Length of data source      */  
    SQLSMALLINT    desclen;        /* Length of dsdescr          */  
    BOOL           found = FALSE;  
    SQLSMALLINT    funcs[100];  
    SQLINTEGER     rgbValue;  
  
    /*  
    * Initialize environment - allocate environment handle.  
    */  
    rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv );  
    rc = SQLAllocHandle( SQL_HANDLE_DBC, hEnv, &hDbc );  
  
    /*  
    * Use SQLDataSources to verify MVSDB2 does exist.  
    */  
    while( ( rc = SQLDataSources( hEnv,  
                                SQL_FETCH_NEXT,  
                                dsname,  
                                SQL_MAX_DSN_LENGTH+1,  
                                &dslen,  
                                dsdescr,  
                                &desclen ) ) != SQL_NO_DATA_FOUND )  
    {  
        if( !strcmp( dsname, "MVSDB2" ) ) /* data source exist */  
        {  
            found = TRUE;  
            break;  
        }  
    }  
  
    if( !found )  
    {  
        fprintf(stdout, "Data source %s does not exist...\n", dsname );  
        fprintf(stdout, "program aborted.\n");  
        exit(1);  
    }  
}
```

```

if( ( rc = SQLConnect( hDbc, dsname, SQL_NTS, "myid", SQL_NTS,
                    "mypd", SQL_NTS ) )== SQL_SUCCESS )
{
    fprintf( stdout, "Connect to %s\n", dsname );
}

SQLAllocHandle( SQL_HANDLE_STMT, hDbc, &hStmt );

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

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

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

```

Chapter 4. Configuring DB2 ODBC and running sample applications

This section provides information about installing DB2 ODBC, the DB2 ODBC runtime environment, and the preparation steps needed to run a DB2 ODBC application.

- “Installing DB2 ODBC”
- “DB2 ODBC runtime environment”
- “Setting up DB2 ODBC runtime environment” on page 43
- “Preparing and executing a DB2 ODBC application” on page 48
- “DB2 ODBC initialization file” on page 52
- “DB2 ODBC migration considerations” on page 65

Installing DB2 ODBC

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

1. Copy and edit the SMP/E jobs.

For sample JCL to invoke the MVS utility IEBCOPY to copy the SMP/E jobs to disk, see the *DB2 Program Directory*.

2. Run the receive job: DSNRECV3.
3. Run the apply job: DSNAPPLY.
4. Run the accept job: DSNACCEP.

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

DB2 ODBC runtime environment

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

The DB2 ODBC driver can use either the call attachment facility (CAF) or the Recoverable Resource Manager Services attachment facility (RRSAF) to connect to the DB2 for OS/390 and z/OS address space.

- If the DB2 ODBC application is not running as a DB2 for OS/390 and 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 OS/390 and z/OS stored procedure, then DB2 ODBC uses the attachment facility that was specified for stored procedures.

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

DB2 ODBC supports access to the local DB2 for OS/390 and z/OS subsystems and any remote data source that is accessible using DB2 for OS/390 and z/OS Version 7. This includes:

- Remote DB2 subsystems using specification of an alias or three-part name
- Remote DRDA-1 and DRDA-2 servers using LU 6.2 or TCP/IP.

The relationship between the application, the DB2 for OS/390 and z/OS V7 ODBC driver and the DB2 for OS/390 and z/OS subsystem are illustrated in Figure 4.

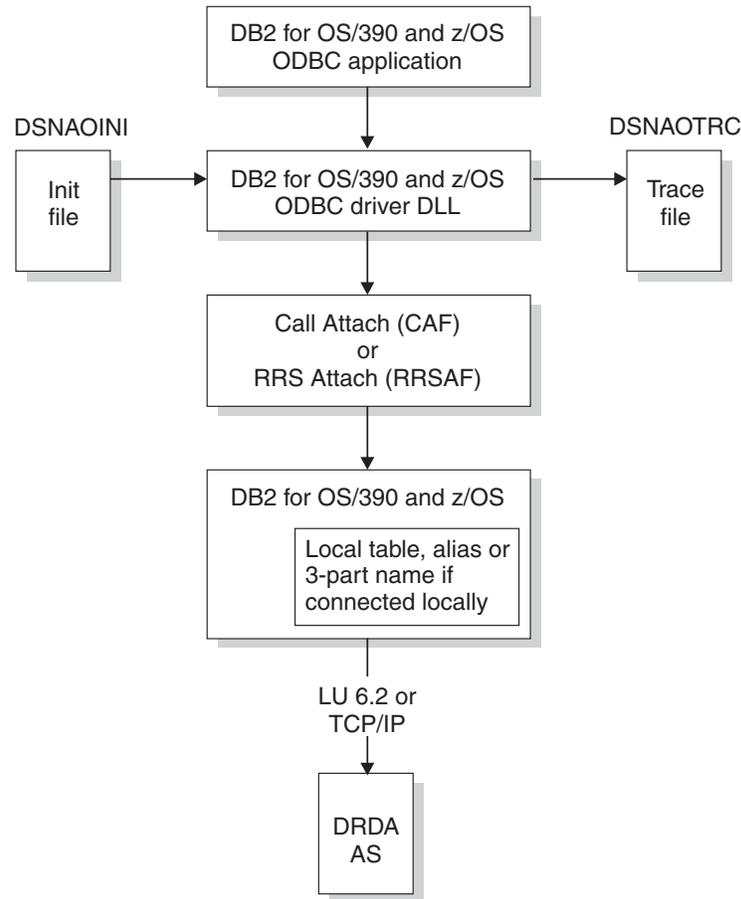


Figure 4. Relationship between DB2 for OS/390 and z/OS V7 ODBC components

Connectivity requirements

DB2 for OS/390 and z/OS V7 ODBC has the following connectivity requirements:

- DB2 ODBC applications must execute on a machine on which Version 7 of DB2 for OS/390 and z/OS is installed.
- If the application is executing with MULTICONTEXT=1, then there are multiple physical connections. Each connection corresponds to an independent transaction and DB2 thread.
- If the application is executing CONNECT (type 1) (described in “Connect type 1 and type 2” on page 18) and MULTICONTEXT=0, then there is only one current physical connection and one transaction on that connection. All transactions on logical connections (that is, with a valid connection handle) are rolled back by the application or committed by DB2 ODBC. This is a deviation from the ODBC connection model.

Setting up DB2 ODBC runtime environment

This section describes the general setup required to enable DB2 ODBC applications. The steps in this section only need to be performed once, and are usually performed as part of the installation process for DB2 for OS/390 and z/OS.

The DB2 ODBC bind files must be bound to the data source. The following two bind steps are required:

- Create packages at every data source
- Create at least one plan to name those packages.

These bind steps are described in the following sections:

- “Bind DBRMs to packages”
- “Bind an application plan” on page 45

The online bind sample, DSNTIJCL, is available in DSN710.SDSNSAMP. We strongly recommend that you use this bind sample as a guide for binding DBRMs to packages and binding an application plan.

Special considerations for the OS/390 UNIX environment are described in the following section:

- “Setting up OS/390 UNIX environment” on page 46

In OS/390 Release 6, the name OS/390 OpenEdition® is replaced with OS/390 UNIX System Services or OS/390 UNIX. Throughout this book, we use the name OS/390 UNIX.

Setup required for using suffix-W APIs is described in the following section:
• “Setting up suffix-W API support” on page 46

Bind DBRMs to packages

This section explains how to bind DBRMs to packages. Use the online bind sample, DSN710.SDSNSAMP(DSNTIJCL), for guidance.

For an application to access a data source using DB2 ODBC, the following IBM supplied DBRMs (shipped in DSN710.SDSNDBRM) must be bound to their corresponding data sources, including the local DB2 for OS/390 and z/OS subsystem and all remote (DRDA) data sources.

• Bind the following DBRMs to all data sources:
– **DSNCLICS** with ISOLATION(CS)*
– **DSNCLIRR** with ISOLATION(RR)*
– **DSNCLIRS** with ISOLATION(RS)*
– **DSNCLIUR** with ISOLATION(UR)*
– **DSNCLINC** with ISOLATION(NC)

Note:

*For DB2 UDB Version 5.2 or earlier, do not use these DBRMs. Bind the
following four DBRMs instead:

- **DSNCLICU** with ISOLATION(CS)
- **DSNCLIRU** with ISOLATION(RR)
- **DSNCLISU** with ISOLATION(RS)
- **DSNCLIUU** with ISOLATION(UR)

- For bind of DSNCLIF4 package to DB2 for OS/390 and z/OS for SYSIBM.LOCATIONS due to differences in catalog table names between releases. For example, when bound to a Version 6 system you receive this warning message:
SYSIBM.SYSLOCATIONS IS NOT DEFINED

Bind packages at remote sites

For an application to access a data source using DB2 ODBC, bind the DBRMs listed above to all data sources, including the local DB2 for OS/390 and z/OS subsystem and all remote (DRDA) data sources. The `SQLConnect()` argument `szDSN` identifies the data source. The data source is the location in the DB2 SYSIBM.LOCATION catalog table. An application running under DB2 ODBC to a remote DB2 for OS/390 and z/OS, or another DBMS, does not need to be bound into the DB2 ODBC plan; rather it can be bound as a package at the remote site. Failure to bind the package at the remote site results in SQLCODE -805.

Bind stored procedures

A stored procedure running under DB2 ODBC to a remote DB2 for OS/390 and z/OS, or another DBMS, does not need to be bound into the DB2 ODBC plan; rather it can be bound as a package at the remote site.

For a stored procedure that resides on the local DB2 for OS/390 and z/OS, the stored procedure package must be bound in the DB2 ODBC plan, using PKLIST. Stored procedures on remote servers only need to bind to that remote server as a package.

For example, DB2 ODBC must always be bound in a plan to a DB2 for OS/390 and z/OS subsystem to which DB2 ODBC first establishes an affinity on the `SQLAllocHandle()` call (with `HandleType` set to `SQL_HANDLE_ENV`). This is the local DB2. The scenario in this example is equivalent to specifying the `MVSDEFAULTSSID` keyword in the initialization file. If DB2 ODBC calls a stored procedure that resides at this local DB2 for OS/390 and z/OS, that stored procedure package must be in the DB2 ODBC plan, using PKLIST.

This process is unique to DB2 for OS/390 and z/OS stored procedure support. For more information about using stored procedures, see “Using stored procedures” on page 417.

Include local, remote, and stored procedure packages in the PKLIST of the plan at the site where the client will execute.

Bind an application plan

This section explains how to bind an application plan. Use the online bind sample, DSN710.SDSNSAMP(DSNTIJCL), for guidance.

A DB2 plan must be created using the PKLIST keyword to name all packages listed in “Bind DBRMs to packages” on page 43. Any name can be selected for the plan; the default name is **DSNACLI**. If a name other than the default is selected, that name must be specified within the initialization file by using the PLANNAME keyword.

Plan bind options

Use PLAN bind options as follows:

- DISCONNECT(EXPLICIT)

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

- CURRENTSERVER

Do not specify this keyword when binding plans.

Setting up OS/390 UNIX environment

To use DB2 ODBC in the OS/390 UNIX environment, the DB2 ODBC definition side-deck must be available to OS/390 UNIX users.

The OS/390 UNIX compiler determines the contents of an input file based on the file extension. In the case of a file residing in an MVS partitioned data set (PDS), the last qualifier in the PDS name is treated as the file extension.

The OS/390 UNIX compiler recognizes the DB2 ODBC definition side-deck by these criteria:

- It must reside in an MVS PDS
- The last qualifier in the PDS name must be .EXP

Therefore, to make the DB2 ODBC definition side-deck available to OS/390 UNIX users, you should define an MVS data set alias that uses .EXP as the last qualifier in the name. This alias should relate to the SDSNMACS data set which is where the DB2 ODBC definition side-deck is installed.

For example, assume that DB2 is installed using DSN710 as the high level data set qualifier. You can define the alias using the following command:

```
DEFINE ALIAS(NAME('DSN710.SDSNC.EXP') RELATE('DSN710.SDSNMACS'))
```

This alias allows OS/390 UNIX users to directly reference the DB2 ODBC definition side-deck by specifying:

```
"//'DSN710.SDSNC.EXP(DSNAOCLI)'"
```

as one of the input files to the OS/390 UNIX c89 command.

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

For the c89 compiler issue:

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

For the cxx compiler issue:

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

Setting up suffix-W API support

This section provides an overview of the setup required to enable Unicode UCS-2 support. DB2 ODBC provides Unicode UCS-2 support with suffix-W APIs. See "Using Unicode functions" on page 429 for information about suffix-W API support.

You need to set up OS/390 support for Unicode which includes a conversion environment and conversion services that use the conversion environment. Installation and customization is required for PTF UQ54199. You must install

OS/390 V2 Release 8, Release 9, Release 10 support for Unicode: FMID HUNI2A0.
After completing the installation, you need to customize the conversion
environment.

If PTF UQ54199 is not applied to your system, installation and customization is not
required.

Installing and activating OS/390 support for Unicode

Follow the instructions in *Program Directory for OS/390 V2 R8/R9/R10 support for*
Unicode to install and activate OS/390 V2 R8/R9/R10 support for Unicode. You
must perform the following tasks:

- # 1. Obtain the SMP/E PTFs UR52471 and UR52473.
- # 2. Download the package: OS/390 V2 R8/R9/R10 support for Unicode.
- # 3. Run the sample job provided in README.TXT.
- # 4. Complete the installation.

Customizing the conversion environment

After installation is complete, follow instructions in *OS/390 Support for Unicode:*
Using Conversion Services to customize the Unicode conversion environment. You
must perform the following tasks:

- # 1. Use the following REXX execs to customize the jobs provided for your
conversion environment:
 - # • CUNRUCST lets you customize job values
 - # • CUNRUALL supplies the values you specify on JCL images
- # 2. Create the conversion environment.
 - # a. Create the conversion image.

Use the image generator in batch job hlq.SCUNJCL(CUNJIUTL) to create
the conversion image. After you create the conversion image, the updated
CUNJIUTL JCL member should appear as shown in Figure 5 on page 48.
#

```

# //CUNMIUTL EXEC PGM=CUNMIUTL
# //SYSPRINT DD SYSOUT=*
# //TABIN DD DISP=SHR,DSN=h1q.SCUNTBL
# //SYSIMG DD DSN=h1q.IMAGES(CUNIMG00),DISP=SHR
# //SYSIN DD *
# /*****
# * INPUT STATEMENTS FOR THE IMAGE GENERATOR *
# *****/
# CONVERSION 00850,01047,ER; /* ASCII -> EBCDIC */
# CONVERSION 01047,00850,ER; /* EBCDIC -> ASCII */
# CONVERSION 00037,1200,ER; /* EBCDIC 037 -> UCS-2 */
# CONVERSION 1200,00037,ER; /* UCS-2 -> EBCDIC 037 */
# CONVERSION 00500,1200,ER; /* Latin-1 EBC -> UCS-2 */
# CONVERSION 1200,00500,ER; /* UCS-2 -> Latin-1 EBC */
# CONVERSION 01047,1200,ER; /* EBCDIC 1047 -> UCS-2 */
# CONVERSION 1200,01047,ER; /* UCS-2 -> EBCDIC 1047 */
# CONVERSION 01208,1200,ER; /* UnicodeCCSID-> UCS-2 */
# CONVERSION 1200,01208,ER; /* UCS-2 -> UnicodeCCSID*/
# CONVERSION 01383,1200,ER; /* Simp Chines -> UCS-2 */
# CONVERSION 1200,01383,ER; /* UCS-2 -> Simp Chines */
# CONVERSION 00932,1200,ER; /* Jpn MCCSID -> UCS-2 */
# CONVERSION 1200,00932,ER; /* UCS-2 -> Jpn MCCSID */
# CONVERSION 00939,1200,ER; /* Jpn-ExtEng -> UCS-2 */
# CONVERSION 1200,00939,ER; /* UCS-2 -> Jpn-ExtEng */
# CONVERSION 00300,1200,ER; /* Jpn GCCSID -> UCS-2 */
# CONVERSION 1200,00300,ER; /* UCS-2 -> Jpn GCCSID */
# CONVERSION 00500,00850,ER; /* Latin-1 EBC -> ASCII */
# CONVERSION 00850,00500,ER; /* ASCII -> Latin-1 EBC */

```

Figure 5. Updated CUNJIUTL JCL member

- b. Calculate the main storage required for a conversion image.
Issue the system command "D UNI,STORAGE" to determine the size of a currently active image.
Find the CUN1017I message in the SYSPRINT log to determine the size of the new conversion image. For example:
CUN1017I GENERATED IMAGE SIZE 291 PAGES.....
 - c. Create parmlib member CUNUNLxx to activate a conversion environment.
 - d. Edit IEASYSxx.
 - e. IPL to initialize the conversion environment.
3. Change or check the status of the conversion environment.
 - Use the DISPLAY UNI system command to view the status of the conversion environment.
 - Use the SET UNI system command to change the conversion environment.

Preparing and executing a DB2 ODBC application

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

Figure 6 on page 49 shows the DB2 ODBC components used to build the DB2 ODBC DLL, and the process you follow to install and prepare a DB2 ODBC application. The shaded areas identify the components that are shipped.

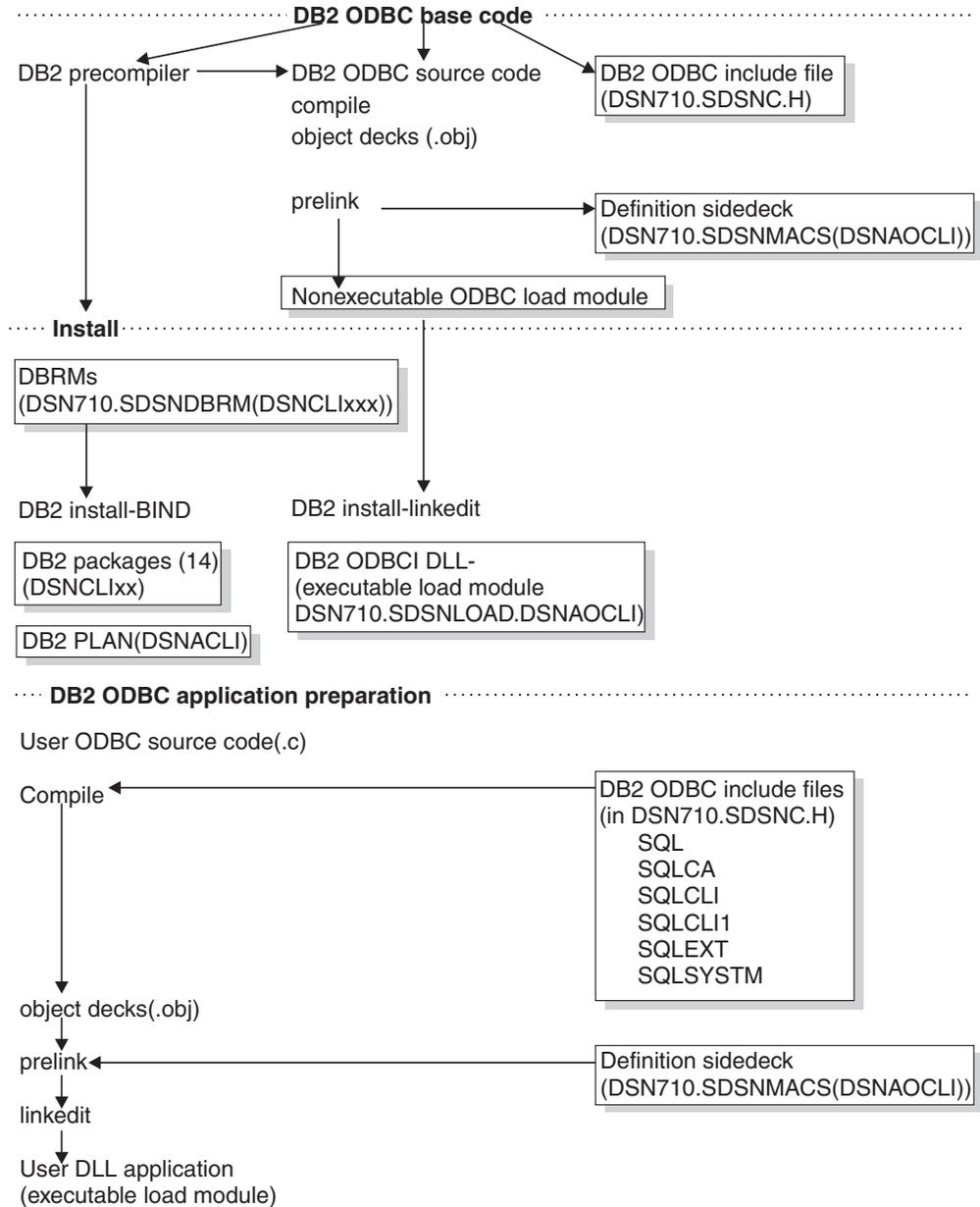


Figure 6. DB2 ODBC application development and execution

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

- “DB2 ODBC application requirements”
- “Application preparation and execution steps” on page 50

DB2 ODBC application requirements

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

The C++ compiler always generates DLL linkage. However, the C compiler only generates DLL linkage if the DLL compile option is used. Failure to generate the

necessary DLL linkage can cause the prelinker and linkage editor to issue warning messages for unresolved references to DB2 ODBC functions.

The minimum requirements for a DB2 ODBC application are as follows:

- OS/390 Version 1 Release 3 Application Enablement optional feature for C/C++.
If the C compiler is used, then the DLL compiler option must be specified.
- OS/390 Release 3 Language Environment Application Enablement base feature.
- The DB2 ODBC application must be written and link-edited to execute with a 31-bit addressing mode, AMODE(31).

Special considerations for OS/390 UNIX

A special consideration applies to DB2 ODBC product data set access. If you build a DB2 ODBC application in OS/390 UNIX, you can use the c89 compile command to compile your application. Although you compile your application under OS/390 UNIX, you can directly reference the non-HFS DB2 ODBC data sets in the c89 command. There is no need to copy the DB2 ODBC product files to HFS.

Application preparation and execution steps

The following steps describe application preparation and execution:

- “Step 1. Compile the application”
- “Step 2. Pre-link and link-edit the application” on page 51
- “Step 3. Execute the application” on page 52

DB2 ODBC provides online samples for installation verification:

DSN803VP

A sample C application. You can use this sample to verify that your DB2 ODBC 3.0 installation is correct. See “DSN803VP sample application” on page 508.

DSN8OIVP

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

DSNTEJ8

Sample JCL. You can use this sample to compile, pre-link, link-edit, and execute the sample application DSN803VP or DSN8OIVP.

The DSNTEJ8, DSN803VP, and DSN8OIVP online samples are available in DSN710.SDSNSAMP. We strongly recommend that you use these samples for guidance when running an application.

Using the ODBC sample in OS/390 UNIX: To use the ODBC sample DSN803VP or DSN8OIVP in OS/390 UNIX, copy DSN803VP or DSN8OIVP from the sample data set to HFS. user/db2 is considered the user’s directory. For example:

```
output 'dsn710.sdsnsamp(dsn803vp)' '/usr/db2/dsn803vp.c' TEXT
```

Step 1. Compile the application

Include the following statement in your DB2 ODBC application:

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

The sqlcli1.h file includes all information that is required for compiling your DB2 ODBC application. All DB2 ODBC header files, including sqlcli1.h, that define the function prototypes, constants, and data structures that are needed for a DB2

ODBC application are shipped in the DSN710.SDSNC.H data set. Therefore, you must add this dataset to your SYSPATH concatenation when you compile your DB2 ODBC application.

For an example of a compile job, use the DSNTEJ8 online sample in DSN710.SDSNSAMP.

Compiling in OS/390 UNIX: If you build a DB2 ODBC application in OS/390 UNIX, you can use the c89 command to compile your application. For example, to compile a C application named 'dsn8o3vp.c' that resides in the current working directory, the c89 compile command might look like:

```
c89 -c -W 'c,d11,long,source,list' -  
-I"//DSN710.SDSNC.H" \  
dsn8o3vp.c
```

Alternatively, if you write an application in C++, the cxx command might look like:

```
cxx -c -W 'c,long,source,list' -  
-I"//DSN710.SDSNC.H" \  
dsn8o3vp.C
```

If your source code is in C, rather than C++, you must compile using the 'd11' option to enable use of the DB2 ODBC driver. This is a requirement even when using the cxx compile command to compile C parts.

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

Before you can link-edit your DB2 ODBC application, you must pre-link your application with the DB2 ODBC definition side-deck provided with Version 7 of DB2 for OS/390 and z/OS.

The definition side-deck defines all of the exported functions in the DB2 ODBC dynamic load library, DSNAOCLI. It resides in the DSN710.SDSNMACS data set, as member DSNAOCLI. The definition side-deck should also be available under the alias data set name of DSN710.SDSNC.EXP as member DSNAOCLI (see “Setting up OS/390 UNIX environment” on page 46 for details). You must include the DSNAOCLI member as input to the Prelinker by specifying it in the pre-link SYSIN DD card concatenation.

For an example of pre-link and link-edit jobs, use the DSNTEJ8 online sample in DSN710.SDSNSAMP.

For more information about DLL, see *OS/390 C/C++ Programming Guide*

Pre-linking and link-editing in OS/390 UNIX: If you build a DB2 ODBC application in OS/390 UNIX, you can use the c89 command to pre-link and link-edit your application. You need to include the DB2 ODBC definition side-deck as one of the input data sets to the c89 command and specify 'd11' as one of the link-edit options.

For example, assume that you have already compiled a C application named 'myapp.c' to create a 'myapp.o' file in the current working directory. The c89 command to pre-link and link-edit your application might look like:

```
c89 -W l,p,map,noer -W l,d11,AMODE=31,map \  
-o dsn8o3vp dsn8o3vp.o "//'DSN710.SDSNC.EXP(DSNAOCLI)'"
```

The following command references the ODBC side-deck as a C89 _XSUFFIX_HOST environmental variable (as described in “Setting up OS/390 UNIX environment” on page 46):

```
c89 -W l,p,map,noer -W l,dll,AMODE=31,map -o dsn8o3vp dsn8o3vp.o
"//DSN710.SDSNMACS(dsnaocli)'"
```

Step 3. Execute the application

DB2 ODBC applications must access the DSN710.SDSNLOAD data set at execution time. The SDSNLOAD data set contains both the DB2 ODBC dynamic load library and the attachment facility used to communicate with DB2.

In addition, the DB2 ODBC driver accesses the DB2 for OS/390 and z/OS load module DSNHDECP. DSNHDECP contains, among other things, the coded character set ID (CCSID) information that DB2 for OS/390 and z/OS uses.

A default DSNHDECP is shipped with DB2 for OS/390 and z/OS in the DSN710.SDSNLOAD data set. However, if the values provided in the default DSNHDECP are not appropriate for your site, a new DSNHDECP can be created during the installation of DB2 for OS/390 and z/OS. If a site specific DSNHDECP is created during installation, you should concatenate the data set containing the new DSNHDECP before the DSN710.SDSNLOAD data set in your STEPLIB or JOBLIB DD card.

For an example of an execute job, use the DSNTTEJ8 online sample in DSN710.SDSNSAMP.

Executing in OS/390 UNIX: To execute a DB2 ODBC application in OS/390 UNIX, you need to include the DSN710.SDSNEXIT and DSN710.SDSNLOAD data sets in the data set concatenation of your STEPLIB environmental variable. The STEPLIB environmental variable can be set in your .profile with the statement:

```
export STEPLIB=DSN710.SDSNEXIT:DSN710.SDSNLOAD
```

Defining a subsystem

There are two ways to define a DB2 subsystem to DB2 ODBC. 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 DSN710.SDSNEXIT data set. In this case, your STEPLIB DD card includes:

```
//STEPLIB DD DSN=DSN710.SDSNEXIT,DISP=SHR
// DD DSN=DSN710.SDSNLOAD,DISP=SHR
...
```

DB2 ODBC initialization file

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

For most applications, use of the DB2 ODBC initialization file is not necessary. However, to make better use of IBM RDBMS features, the keywords can be specified to:

- Help improve the performance or usability of an application.
- Provide support for applications written for a previous version of DB2 ODBC.
- Provide specific work-arounds for existing ODBC applications.

The following sections describe how to create the initialization file and define the keywords:

- “Using the initialization file”
- “Initialization keywords” on page 55

Using the initialization file

The DB2 ODBC initialization file is read at application run time. The file can be specified by either a DSNAOINI DD card or by defining a DSNAOINI OS/390 UNIX environmental variable. DB2 ODBC opens the DSNAOINI data set allocated in your JCL first. If a DSNAOINI data set is not allocated, then DB2 ODBC opens the environmental variable data set.

The initialization file specified can be either a tradition MVS data set or an OS/390 UNIX HFS file. For MVS data sets, the record format of the initialization file can be either fixed or variable length.

The following JCL examples use a DSNAOINI JCL DD card to specify the DB2 ODBC initialization file types supported:

MVS sequential data set USER1.DB2ODBC.ODBCINI:

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

MVS partitioned data set USER1.DB2ODBC.DATA, member ODBCINI:

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

Inline JCL DSNAOINI DD specification:

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

HFS file /u/user1/db2odbc/odbcini:

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

The following examples of OS/390 UNIX export statements define the DB2 ODBC DSNAOINI OS/390 UNIX environmental variable for the DB2 ODBC initialization file types supported:

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

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

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

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

MVS sequential data set USER1.ODBCINI:

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

Redirecting to use a file specified by another DD card, MYDD, that is already allocated:

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

MVS partitioned data set USER1.DB2ODBC.DATA, member ODBCINI:

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

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

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

Initialization file structure

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

Common section

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

Subsystem section

Contains parameter values unique to that subsystem.

Data source sections

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

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

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

```
[data-source-name]
```

This is the *section header*.

The parameters are set by specifying a keyword with its associated keyword value in the form:

```
KeywordName =keywordValue
```

- All the keywords and their associated values for each data source must be located below the data source section header.
- The keyword settings in each section apply only to the data source name in that section header.
- The keywords are **not** case sensitive; however, their values can be if the values are character based.
- For the syntax associated with each keyword, see "Initialization keywords" on page 55.
- 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 semi-colon 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).
- **Attention:** 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 is a sample 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=V61A

; Example SUBSYSTEM stanza for V61A subsystem
[V61A]
MVSATTACHTYPE=CAF
PLANNAME=DSNACLI

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

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

Initialization keywords

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

APPLTRACE = 0 | 1

This keyword is placed in the common section.

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

- 0 = Disabled (default)
- 1 = Enabled

For more information about using the APPLTRACE keyword, see “Application trace” on page 457.

Important: This keyword is renamed. DB2 ignores the Version 6 keyword name CLITRACE.

APPLTRACEFILENAME = *dataset name*

This keyword is placed in the common section.

APPLTRACEFILENAME is only used if a trace is started by the APPLTRACE keyword. When APPLTRACE is set to 1, use the APPLTRACEFILENAME keyword to identify an MVS data set name or OS/390 UNIX HFS file name that records the DB2 ODBC application trace. “Diagnostic trace” on page 459 provides detailed information about specifying file name formats.

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

AUTOCOMMIT = 1 | 0

This keyword is placed in the data source section.

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

1 = on (default)

0 = off

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

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

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

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

BITDATA = 1 | 0

This keyword is placed in the data source section.

The BITDATA keyword allows you to specify whether ODBC binary data types, SQL_BINARY, SQL_VARBINARY, and SQL_LONGVARBINARY, and SQL_BLOB are reported as binary type data. IBM DBMSs support columns with binary data types by defining CHAR, VARCHAR and LONG VARCHAR columns with the FOR BIT DATA attribute.

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

1 = Report FOR BIT DATA and BLOB data types as binary data types. This is the default.

0 = Disabled.

CLISCHEMA = *schema_name*

This keyword is placed in the data source section.

The CLISCHEMA keyword lets you indicate the schema of the DB2 ODBC shadow catalog tables or views to search when you issue an ODBC catalog function call. For example, if you specify CLISCHEMA=PAYROLL, the ODBC catalog functions that normally reference the DB2 system catalog tables (SYSIBM schema), will reference the following views of the DB2 ODBC shadow catalog tables:

- PAYROLL.COLUMNS
- PAYROLL.TABLES

- PAYROLL.COLUMNPRIVILIGES
- PAYROLL.TABLEPRIVILIGES
- PAYROLL.SPECIALCOLUMNS
- PAYROLL.PRIMARYKEYS
- PAYROLL.FOREIGNKEYS
- PAYROLL.TSTATISTICS
- PAYROLL.PROCEDURES

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

COLLECTIONID = *collection_id*

This keyword is placed in the data source section.

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

The value is a character string and must not exceed 18 characters. It can be overridden by executing the SET CURRENT PACKAGESET statement.

CONNECTTYPE = 1 | 2

This keyword is placed in the common section.

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

- 1 = Multiple concurrent connections, each with its own commit scope. If MULTICONTEXT=0 is specified, a new connection might not be added unless the current transaction on the current connection is on a transaction boundary (either committed or rolled back). This is the default.
- 2 = Coordinated connections where multiple data sources participate under the same distributed unit of work. CONNECTTYPE=2 is ignored if MULTICONTEXT=1 is specified.

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

This keyword is placed in the data source section.

The CURRENTFUNCTIONPATH keyword defines the path used to resolve unqualified user-defined functions, distinct types, and stored procedure references that are used in dynamic SQL statements. It contains a list of one or more schema names, which are used to set the CURRENT PATH special register using the SET CURRENT PATH SQL statement upon connection to the data source. Each schema name in the keyword string must be delimited with single quotes and separated by commas. The entire keyword string must be enclosed in double quotes and must not exceed 254 characters.

The default value of the CURRENT PATH special register is:

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

where 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, then it is implicitly assumed at the beginning of the path in the order shown above. The order of the schema names in the path determines the order in which the names are resolved. For more detailed information on schema name resolution, see *DB2 SQL Reference*.

Unqualified user-defined functions, distinct types, and stored procedures are searched from the list of schemas specified in the CURRENTFUNCTIONPATH

setting in the order specified. If the user-defined function, distinct type, or stored procedures is not found in a specified schema, the search continues in the schema specified next in the list. For example:

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

results in searching schema "USER01", followed by schema "PAYROLL", followed by schema "SYSIBM", etc..

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

CURRENTSQLID = *current_sqlid*

This keyword is placed in the data source section.

The CURRENTSQLID keyword is valid only for those DB2 DBMSs that support SET CURRENT SQLID (such as DB2 for OS/390 and z/OS). If this keyword is present, then a SET CURRENT SQLID statement is sent to the DBMS after a successful connect. This allows the end user and the application to name SQL objects without having to qualify by schema name.

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.

This keyword can be used by an end user to improve performance. If the user is sure that the application:

1. Does not have behavior that is dependent on the SQL_CURSOR_COMMIT_BEHAVIOR or the SQL_CURSOR_ROLLBACK_BEHAVIOR information returned using SQLGetInfo(), and
2. Does not require cursors to be preserved from one transaction to the next,

then the value of this keyword can be set to **0**. The DBMS operates more efficiently as resources no longer need to be maintained after the end of a transaction.

DBNAME = *dbname*

This keyword is placed in the data source section.

The DBNAME keyword is only used when connecting to DB2 for OS/390 and z/OS, and only if (*base*) table catalog information is requested by the application.

If a large number of tables exist in the DB2 for OS/390 and z/OS subsystem, a *dbname* can be specified to reduce the time it takes for the database to process the catalog query for table information, and reduce the number of tables returned to the application.

The value of the *dbname* keyword maps to the DBNAME column in the DB2 for OS/390 and z/OS system catalog tables. If no value is specified, or if views, synonyms, system tables, or aliases are also specified using TABLETYPE, only table information is restricted; views, aliases, and synonyms are not restricted with DBNAME. This keyword can be used in conjunction with SCHEMALIST and TABLETYPE to further limit the number of tables for which information is returned.

DIAGTRACE = 0 | 1

This keyword is placed in the common section.

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

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

You can enable the diagnostic trace using the DSNAOTRC command when the DIAGTRACE keyword is set to 0.

1 = The DB2 ODBC diagnostic trace is enabled. Diagnostic data is recorded in the application address space. If you include a DSNAOTRC DD statement in your job or TSO logon procedure that identifies an MVS data set or an OS/390 UNIX HFS file name, the trace is externalized at normal program termination. You can format the trace using the DSNAOTRC trace formatting program.

For more information about using the DIAGTRACE keyword and the DSNAOTRC command, see “Diagnostic trace” on page 459.

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

DIAGTRACE_BUFFER_SIZE = *buffer size*

This keyword is placed in the common section.

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

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

If a trace is already active, this keyword is ignored.

Important: DB2 ignores the Version 6 keyword name TRACE_BUFFER_SIZE.

DIAGTRACE_NO_WRAP = 0 | 1

This keyword is placed in the common section.

The DIAGTRACE_NO_WRAP keyword controls the behavior of the DB2 ODBC diagnostic trace when the DB2 ODBC diagnostic trace buffer fills up. This keyword is only used if a trace is started by the DIAGTRACE keyword.

0 = The trace table is a wrap-around 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 already active, this keyword is ignored.

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

GRAPHIC = 0 | 1 | 2 | 3

This keyword is placed in the data source section.

The GRAPHIC keyword controls whether DB2 ODBC reports IBM GRAPHIC (double byte character support) as one of the supported data types when SQLGetTypeInfo() is called. SQLGetTypeInfo() lists the data types supported by the data source for the current connection. These are not native ODBC types but have been added to expose these types to an application connected to a DB2 family product.

0 = disabled (default)

1 = enabled

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

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

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

MAXCONN = 0 | positive number

This keyword is placed in the common section.

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

A value of 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.

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. There is only one physical connection to either the local DB2/MVS subsystem or a remote DB2 subsystem or remote DRDA-1 or DRDA-2 server.

MULTICONTEXT = 0 | 1

This keyword is placed in the common section.

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

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

ODBC external context management support requires MULTICONTEXT=0, MVSATTACHTYPE=RRSAF, and OS/390 Version 2 Release 5 or higher. DB2 ODBC does not support external contexts when the application runs under a stored procedure.

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

1 = The DB2 ODBC code creates an independent context for a data source connection at the connection handle level when `SQLAllocHandle()` is issued. Each connection to multiple data sources is governed by `CONNECTTYPE=1` rules and is associated with an independent DB2 thread. Connection switching among multiple data sources is not prevented due to the commit status of the transaction; an application can use multiple connection handles without having to perform a commit or rollback on a connection before switching to another connection handle. The use of `MULTICONTEXT=1` requires `MVSATTACHTYPE=RRSAF` and OS/390 Version 2 Release 5 or higher.

The application can use `SQLGetInfo()` with `infoType=SQL_MULTIPLE_ACTIVE_TXN` to determine whether `MULTICONTEXT=1` is supported.

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

- The application created a DB2 thread before invoking DB2 ODBC. This is always the case for a stored procedure using DB2 ODBC.
- The application created and switched to a private context using OS/390 Context Services before invoking DB2 ODBC.
- The application started a unit of recovery with any RRS resource manager (for example, IMS) before invoking DB2 ODBC.
- `MVSATTACHTYPE=CAF` is specified in the initialization file.
- The OS/390 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 OS/390 and z/OS attachment type that DB2 ODBC uses to connect to the DB2 for OS/390 and z/OS address space. This parameter is ignored if the DB2 ODBC application is running as a DB2 for OS/390 and z/OS stored procedure. In that case, DB2 ODBC uses the attachment type that was defined for the stored procedure.

CAF: DB2 ODBC uses the DB2 for OS/390 and z/OS call attachment facility (CAF). This is the default.

RRSAF: DB2 ODBC uses the DB2 for OS/390 and z/OS Recoverable Resource Manager Services attachment facility (RRSAF).

ODBC external context management support requires `MVSATTACHTYPE=RRSAF`, `MULTICONTEXT=0`, and OS/390 Version 2 Release 5 or higher. DB2 ODBC does not support external contexts when the application runs under a stored procedure.

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

MVSDEFAULTSSID = ssid

This keyword is placed in the common section.

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

OPTIMIZEFORNROWS = integer

This keyword is placed in the data source section.

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

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

PATCH2 = patch number

This keyword is placed in the data source section.

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

- 0: No workaround (default).
- 300

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

Table 9 explains how PATCH2 settings affect return codes.

Table 9. PATCH2 settings and SQL return codes

If the SQL statement is...	SQLExecute and SQLExecDirect return...
A searched update or searched delete and no rows satisfy the search condition	<ul style="list-style-type: none"> • SQL_SUCCESS without a patch (PATCH2=0) • SQL_NO_DATA_FOUND with a patch (PATCH2=300)
A mass delete or update and no rows satisfy the search condition	<ul style="list-style-type: none"> • SQL_SUCCESS_WITH_INFO without a patch (PATCH2=0) • SQL_NO_DATA_FOUND with a patch (PATCH2=300)
A mass delete or update and one or more rows satisfy the search condition	SQL_SUCCESS_WITH_INFO without a patch (PATCH2=0) or with a patch (PATCH2=300)

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

PLANNAME = planname

This keyword is placed in the subsystem section.

The PLANNAME keyword specifies the name of the DB2 for OS/390 and z/OS PLAN that was created during installation. A PLAN name is required when initializing the application connection to the DB2 for OS/390 and z/OS subsystem which occurs during the processing of the SQLAllocHandle() call (with HandleType set to SQL_HANDLE_ENV).

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

SCHEMALIST = "schema1', 'schema2' ,..."

This keyword is placed in the data source section.

The SCHEMALIST keyword specifies a list of schemas in the data source.

If there are a large number of tables defined in the database, a schema list can be specified to reduce the time it takes for the application to query table information, and reduce the number of tables listed by the application. Each schema name is case sensitive, must be delimited with single quotes and separated by commas. The entire string must also be enclosed in double quotes, for example:

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

For DB2 for OS/390 and z/OS, CURRENT SQLID can also be included in this list, but without the single quotes, for example:

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

The maximum length of the keyword string is 256 characters.

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

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

SYSSCHEMA = *sysschema*

This keyword is placed in the data source section. This keyword is placed in the data source section.

The 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 system catalog information.

Using this schema name, the system administrator can define a set of views consisting of a subset of the rows for each of the following system catalog tables:

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

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

Defining and using limited views of the system catalog tables reduces the number of tables listed by the application, which reduces the time it takes for the application to query table information.

If no value is specified, the default is:

- SYSIBM on DB2 for OS/390 and z/OS and OS/400
- SYSTEM on DB2 for VSE & VM

- QSYS2 on DB2 for AS/400

This keyword can be used in conjunction with SCHEMALIST, TABLETYPE (and DBNAME on DB2 for OS/390 and z/OS) to further limit the number of tables for which information is returned.

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

This keyword is placed in the data source section.

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

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

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

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

TABLETYPE is used to provide a default for the DB2 ODBC function that retrieves the list of tables, views, aliases, and synonyms in the data source. If the application does not specify a table type on the function call, and this keyword is not used, information about all table types is returned. If the application does supply a value for the *tabletype* on the function call, then that argument value overrides this keyword value.

If TABLETYPE includes any value other than TABLE, then the DBNAME keyword setting cannot be used to restrict information to a particular DB2 for OS/390 and z/OS subsystem.

THREADSAFE= 1 | 0

This keyword is placed in the common section.

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

- 1 = The DB2 ODBC code is threadsafe if the application is executing in a POSIX(ON) environment. Multiple LE threads in the process can use DB2 ODBC. The threadsafe capability cannot be provided in a POSIX(OFF) environment. This is the default.
- 0 = The DB2 ODBC code is not threadsafe. This reduces the overhead of serialization code in DB2 ODBC for applications that are not *multithreaded*, but provides no protection for concurrent LE threads in applications that are multithreaded.

TXNISOLATION = 1 | 2 | 4 | 8 | 32

This keyword is placed in the data source section.

The TXNISOLATION keyword sets the isolation level to:

- 1 = Read uncommitted (uncommitted read)
- 2 = Read committed (cursor stability) (default)
- 4 = Repeatable read (read stability)
- 8 = Serializable (repeatable read)
- 32 = (No commit, DB2 for OS/400 only)

The words in round brackets are the DB2 equivalents for SQL92 isolation levels. Note that *no commit* is not an SQL92 isolation level and is supported only on DATABASE 2 for OS/400. See *DB2 Application Programming and SQL Guide* for more information on isolation levels.

UNDERSCORE = 1 | 0

This keyword is placed in the data source section.

The UNDERSCORE keyword specifies whether the underscore character "_" is to be used as a wildcard character (matching any one character, including no character), or to be used as itself. This parameter only affects catalog function calls that accept search pattern strings.

1 = "_" acts as a wildcard (default)

The underscore is treated as a wildcard matching any one character or none. For example, if two tables are defined as follows:

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

The DB2 ODBC catalog function call that returns table information (SQLTables()) returns both of these entries if "KEY_WORDS" is specified in the table name search pattern argument.

0 = "_" acts as itself

The underscore is treated as itself. If two tables are defined as shown in the example above, SQLTables() returns only the "KEY_WORDS" entry if "KEY_WORDS" is specified in the table name search pattern argument.

Setting this keyword to 0 can result in performance improvement in those cases where object names (owner, table, column) in the data source contain underscores.

DB2 ODBC migration considerations

#

Changes to the columns of the result set for several DB2 ODBC catalog functions are introduced with the DB2 for OS/390 Version 6 ODBC driver. These changes, which include new and reordered columns, will affect existing applications that are migrating from a DB2 for OS/390 Version 5 ODBC driver to a Version 6 ODBC driver *and* are issuing the SQLColumns(), SQLForeignKeys(), or SQLProcedureColumns() APIs. For a list of the columns returned for each of these APIs, see:

- Table 38 on page 125
- Table 63 on page 182
- Table 123 on page 314

Chapter 5. Functions

This section provides a description of each function. Each description has the following sections.

- Purpose
- Syntax
- Function arguments
- Usage
- Return codes
- Diagnostics
- Restrictions
- Example
- References

Each section is described below.

Purpose

This section gives a brief overview of what the function does. It also indicates if any functions should be called before and after calling the function being described.

Each function also has a table, such as the one below that indicates which specification or standard the function conforms to. The first column indicates which version (1.0, 2.0, or 3.0) of the ODBC specification the function was first provided. The second and third columns indicate if the function is included in the X/Open CLI CAE specification and the ISO CLI standard.

Table 10. Sample function specification table

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

This table indicates support of the function. Some functions use a set of options that do not apply to all specifications or standards. The restrictions section identifies any significant differences.

Syntax

This section contains the generic 'C' prototype. If the function is defined by ODBC V2.0, then the prototype should be identical to that specified in *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*.

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

Function arguments

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

Only `SQLGetInfo()` and `SQLBindParameter()` have parameters that are both input and output.

Some functions contain input or output arguments which are known as *deferred* or *bound* arguments. These arguments are pointers to buffers allocated by the application, and are associated with (or bound to) either a parameter in an SQL statement, or a column in a result set. The data areas specified by the function are accessed by DB2 ODBC at a later time. It is important that these deferred data areas are still valid at the time DB2 ODBC accesses them.

Usage

This section provides information about how to use the function, and any special considerations. Possible error conditions are not discussed here, but are listed in the diagnostics section instead.

Return codes

This section lists all the possible function return codes. When `SQL_ERROR` or `SQL_SUCCESS_WITH_INFO` is returned, error information can be obtained by calling `SQLGetDiagRec()`.

See “Diagnostics” on page 28 for more information about return codes.

Diagnostics

This section contains a table that lists the `SQLSTATE`s explicitly returned by DB2 ODBC (`SQLSTATE`s generated by the DBMS can also be returned) and indicates the cause of the error. These values are obtained by calling `SQLGetDiagRec()` after the function returns an `SQL_ERROR` or `SQL_SUCCESS_WITH_INFO`.

See “Diagnostics” on page 28 for more information about diagnostics.

Restrictions

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

Example

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

See Chapter 4, “Configuring DB2 ODBC and running sample applications”, on page 41 for more information on setting up the DB2 ODBC environment and accessing the sample applications.

References

This section lists related DB2 ODBC functions.

Function summary

Table 11 provides a complete list of functions that DB2 ODBC and Microsoft ODBC 3.0 support. For each function, the table indicates whether ODBC or DB2 ODBC supports it, the ODBC conformance level, and a brief description of the function.

Table 11. Function list by category

Task function name	ODBC 3.0	DB2 ODBC support	Purpose
Note: Depr in the ODBC 3.0 column indicates that the function has been deprecated in ODBC. See “DB2 ODBC migration considerations” on page 65 for more information.			
Connecting to a data source			
SQLAllocEnv	Depr	Yes	Obtains an environment handle. One environment handle is used for one or more connections.
SQLAllocConnect	Depr	Yes	Obtains a connection handle.
SQLAllocHandle	Core	Yes	Obtains a handle.
SQLConnect	Core	Yes	Connects to specific driver by data source name, user ID, and password.

Table 11. Function list by category (continued)

Task function name	ODBC 3.0	DB2 ODBC support	Purpose
SQLDriverConnect	Lvl 1	Yes	Connects to a specific driver by connection string or requests that the driver manager and driver display connection dialogs for the user. Note: This function is also extended by the additional IBM keywords supported in the ODBC.INI file in the DB2 UDB CLI environment. Within the DB2 for OS/390 and z/OS CLI environment, there is no equivalent of the ODBC.INI file.
SQLSetConnection	No	Yes	Connects to a specific data source by connection string.
SQLBrowseConnect	Lvl 2	No	Returns successive levels of connection attributes and valid attribute values. When a value is specified for each connection attribute, connects to the data source.
Obtaining information about a driver and data source			
SQLDataSources	Lvl 2	Yes	Returns the list of available data sources.
SQLDrivers	Lvl 2	No	Returns the list of installed drivers and their attributes (ODBC 2.0). Note: This function is implemented within the ODBC driver manager and is therefore not applicable within the DB2 for OS/390 and z/OS ODBC environment.
SQLGetInfo	Lvl 1	Yes	Returns information about a specific driver and data source.
SQLGetFunctions	Lvl 1	Yes	Returns supported driver functions.
SQLGetTypeInfo	Lvl 1	Yes	Returns information about supported data types.
Setting and retrieving driver options			
SQLSetConnectAttr	Core	Yes	Sets a connection attribute.
SQLGetConnectAttr	Core	Yes	Returns the value of a connection attribute.
SQLSetEnvAttr	No	Yes	Sets an environment option.
SQLGetEnvAttr	No	Yes	Returns the value of an environment option.
SQLSetStmtAttr	Core	Yes	Sets a statement attribute.
SQLGetStmtAttr	Core	Yes	Returns the value of a statement attribute.
SQLSetConnectOption	Depr	Yes	Sets a connection option.
SQLGetConnectOption	Depr	Yes	Returns the value of a connection option.
SQLSetStmtOption	Depr	Yes	Sets a statement option.
SQLGetStmtOption	Depr	Yes	Returns the value of a statement option.
Preparing SQL Requests			
SQLAllocStmt	Depr	Yes	Allocates a statement handle.
SQLPrepare	Core	Yes	Prepares an SQL statement for later execution.
SQLBindParameter	Lvl 1	Yes	Assigns storage for a parameter in an SQL statement (ODBC 2.0)
SQLSetParam	Core	Yes	Assigns storage for a parameter in an SQL statement (ODBC 2.0). In ODBC, SQLBindParameter() replaces this function.
SQLParamOptions	Lvl 2	Yes	Specifies the use of multiple values for parameters.

Table 11. Function list by category (continued)

Task function name	ODBC 3.0	DB2 ODBC support	Purpose
SQLGetCursorName	Core	Yes	Returns the cursor name associated with a statement handle.
SQLSetCursorName	Core	Yes	Specifies a cursor name.
SQLSetScrollOptions	Lvl 2	No	Sets options that control cursor behavior.
Submitting Requests			
SQLExecute	Core	Yes	Executes a prepared statement.
SQLExecDirect	Core	Yes	Executes a statement.
SQLNativeSql	Lvl 2	Yes	Returns the text of an SQL statement as translated by the driver.
SQLDescribeParam ^a	Lvl 2	Yes	Returns the description for a specific input parameter in a statement.
SQLNumParams	Lvl 2	Yes	Returns the number of parameters in a statement.
SQLParamData	Lvl 1	Yes	Used in conjunction with SQLPutData() to supply parameter data at execution time. (Useful for long data values.)
SQLPutData	Lvl 1	Yes	Send part or all of a data value for a parameter. (Useful for long data values.)
Retrieving results and information about results			
SQLRowCount	Core	Yes	Returns the number of rows affected by an insert, update, or delete request.
SQLNumResultCols	Core	Yes	Returns the number of columns in the result set.
SQLDescribeCol	Core	Yes	Describes a column in the result set.
SQLColAttributes	Depr	Yes	Describes attributes of a column in the result set.
SQLSetColAttributes	No	Yes	Sets attributes of a column in the result set.
SQLBindCol	Core	Yes	Assigns storage for a result column and specifies the data type.
SQLFetch	Core	Yes	Returns a result row.
SQLExtendedFetch	Lvl 2	Yes	Returns multiple result rows.
SQLGetData	Lvl 1	Yes	Returns part or all of one column of one row of a result set. (Useful for long data values.)
SQLSetPos	Lvl 2	No	Positions a cursor within a fetched block of data.
SQLMoreResults	Lvl 2	Yes	Determines whether there are more result sets available and, if so, initializes processing for the next result set.
SQLError	Depr	Yes	Returns additional error or status information.
SQLGetDiagRec	Core	Yes	Returns additional diagnostic information.
SQLGetSQLCA	No	Yes	Returns the SQLCA associated with a statement handle.
Large object support			
SQLGetLength ^a	No	Yes	Gets length of a string referenced by a LOB locator.
SQLGetPosition ^a	No	Yes	Gets the position of a string within a source string referenced by a LOB locator.

Table 11. Function list by category (continued)

Task function name	ODBC 3.0	DB2 ODBC support	Purpose
SQLGetSubString ^a	No	Yes	Creates a new LOB locator that references a substring within a source string (the source string is also represented by a LOB locator).
Obtaining information about the data source's system tables (catalog functions)			
SQLColumnPrivileges	Lvl 2	Yes	Returns a list of columns and associated privileges for a table.
SQLColumns	Lvl 1	Yes	Returns the list of column names in specified tables.
SQLForeignKeys	Lvl 2	Yes	Returns a list of column names that comprise foreign keys, if they exist for a specified table.
SQLPrimaryKeys	Lvl 2	Yes	Returns the list of column names that comprise the primary key for a table.
SQLProcedureColumns	Lvl 2	Yes	Returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures.
SQLProcedures	Lvl 2	Yes	Returns the list of procedure names stored in a specific data source.
SQLSpecialColumns	Lvl 1	Yes	Returns information about the optimal set of columns that uniquely identifies a row in a specified table, or the columns that are automatically updated when any value in the row is updated by a transaction.
SQLStatistics	Lvl 1	Yes	Returns statistics about a single table and the list of indexes associated with the table.
SQLTablePrivileges	Lvl 2	Yes	Returns a list of tables and the privileges associated with each table.
SQLTables	Lvl 1	Yes	Returns the list of table names stored in a specific data source.
Terminating a Statement			
SQLFreeStmt	Core	Yes	End statement processing and closes the associated cursor, discards pending results, and, optionally, frees all resources associated with the statement handle.
SQLCloseCursor	Core	Yes	Closes a cursor that has been opened on a statement handle.
SQLCancel	Core	Yes	Cancels an SQL statement.
SQLTransact	Depr	Yes	Commits or rolls back a transaction.
SQLEndTran	Core	Yes	Commits or rolls back a transaction.
Terminating a Connection			
SQLDisconnect	Core	Yes	Closes the connection.
SQLFreeHandle	Core	Yes	Releases and environment, connection, statement, or descriptor handle.
SQLFreeConnect	Depr	Yes	Releases the connection handle.
SQLFreeEnv	Depr	Yes	Releases the environment handle.

Table 11. Function list by category (continued)

Task function name	ODBC 3.0	DB2 ODBC support	Purpose
-------------------------------	---------------------	-----------------------------	----------------

Note:

^a This function is supported by Version 6 of DB2 ODBC or later.

ODBC functions not supported by DB2 ODBC:

- `SQLSetPos()` and `SQLBrowseConnect()` are not supported by Call Level Interface or DB2 ODBC.
- `SQLSetScrollOptions()` is not supported. It is superceded by the `SQL_CURSOR_TYPE`, `SQL_CONCURRENCY`, `SQL_KEYSET_SIZE`, and `SQL_ROWSET_SIZE` statement options.
- `SQLDrivers()` is implemented by the ODBC driver manager and is not supported by DB2 ODBC.

SQLAllocConnect - Allocate connection handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

In ODBC 3.0, `SQLAllocHandle()` replaces the ODBC 2.0 function `SQLAllocConnect()`. See `SQLAllocHandle()` for more information.

`SQLAllocConnect()` allocates a connection handle and associated resources within the environment identified by the input environment handle. Call `SQLGetInfo()` with `fInfoType` set to `SQL_ACTIVE_CONNECTIONS`, to query the number of connections that can be allocated at any one time.

While this API is active, the DB2 ODBC driver establishes an affinity with the DB2 subsystem. Processing includes allocating a DB2 for OS/390 and z/OS plan as a resource.

`SQLAllocEnv()` must be called before calling this function.

This function must be called before calling `SQLConnect()` or `SQLDriverConnect()`.

Syntax

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

Function arguments

Table 12. *SQLAllocConnect* arguments

Data type	Argument	Use	Description
SQLHENV	<i>henv</i>	input	Environment handle
SQLHDBC *	<i>phdbc</i>	output	Pointer to connection handle

Usage

The output connection handle is used by DB2 ODBC to reference all information related to the connection, including general status information, transaction state, and error information.

If the pointer to the connection handle (*phdbc*) already points to a valid connection handle previously allocated by `SQLAllocConnect()`, then the original value is overwritten as a result of this call. This is an application programming error which is not detected by DB2 ODBC.

Return codes

- `SQL_SUCCESS`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`

If `SQL_ERROR` is returned, the *phdbc* argument is set to `SQL_NULL_HDBC`. The application should call `SQLGetDiagRec()` with the environment handle (*henv*) and with *hdbc* and *hstmt* arguments set to `SQL_NULL_HDBC` and `SQL_NULL_HSTMT` respectively.

Diagnostics

Table 13. SQLAllocConnect SQLSTATEs

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	This could be a failure to establish the association with the DB2 for OS/390 and z/OS subsystem or any other system related error.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	<i>phdbc</i> was a null pointer.
S1013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
S1014	No more handles.	Returned if the MAXCONN keyword or SQL_MAXCONN attribute is set to a positive integer and the number of connections has reached that value. If MAXCONN is set to zero, there is no limit. DB2 ODBC is not able to allocate a handle due to internal resources.

Restrictions

None.

Example

The following example shows a basic connect, with minimal error handling.

```

/* ... */
/*****
** - demonstrate basic connection to two data sources.
** - error handling mostly ignored for simplicity
**
** Functions used:
**
**   SQLAllocConnect  SQLDisconnect
**   SQLAllocEnv     SQLFreeConnect
**   SQLConnect      SQLFreeEnv
** 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];
    SERVER           svr[MAX_CONNECTIONS] =
        {
            "KARACHI" ,
            "DAMASSUSS"
        }

    /* allocate an environment handle */
    SQLAllocEnv(&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, etc. */
    /****** End Processing Step *****/

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

    SQLTransact (henv,
                 hdbc[0],
                 SQL_COMMIT);

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

    SQLTransact (henv,
                 hdbc[1],
                 SQL_COMMIT);

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

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

    SQLDisconnect(hdbc[1]);    /* disconnect second connection */
    SQLFreeConnect(hdbc[0]);   /* free first connection handle */
    SQLFreeConnect(hdbc[1]);   /* free second connection handle */
    SQLFreeEnv(henv);         /* free environment handle */

    return (SQL_SUCCESS);
}

```

SQLAllocConnect

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

References

- “SQLAllocEnv - Allocate environment handle” on page 77
- “SQLConnect - Connect to a data source” on page 129
- “SQLDriverConnect - (Expanded) connect to a data source” on page 146
- “SQLDisconnect - Disconnect from a data source” on page 144
- “SQLFreeConnect - Free connection handle” on page 189
- “SQLGetConnectOption - Returns current setting of a connect option” on page 202
- “SQLSetConnectOption - Set connection option” on page 345

SQLAllocEnv - Allocate environment handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

In ODBC 3.0, `SQLAllocHandle()` replaces the ODBC 2.0 function `SQLAllocEnv()`. See `SQLAllocHandle()` for more information.

`SQLAllocEnv()` allocates an environment handle and associated resources. There can be only one environment active at any one time per application.

An application must call this function prior to `SQLAllocConnect()` or any other DB2 ODBC functions. The *henv* value is passed in all subsequent function calls that require an environment handle as input.

Syntax

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

Function arguments

Table 14. *SQLAllocEnv* arguments

Data type	Argument	Use	Description
SQLHENV *	<i>phenv</i>	output	Pointer to environment handle

Usage

There can be only one active environment at a time per application. Any subsequent calls to `SQLAllocEnv()` return the same handle as the first `SQLAllocEnv()` call.

`SQLFreeEnv()` must be called for each successful `SQLAllocEnv()` call before the resources associated with the handle are released. `SQLFreeEnv()` must also be called to free a restricted environment handle as described under 'Return Codes' below.

Return codes

- `SQL_SUCCESS`
- `SQL_ERROR`

If `SQL_ERROR` is returned and *phenv* is equal to `SQL_NULL_HENV`, then `SQLGetDiagRec()` cannot be called because there is no handle with which to associate additional diagnostic information.

If the return code is `SQL_ERROR` and the pointer to the environment handle is not equal to `SQL_NULL_HENV`, then the handle is a *restricted handle*. This means the handle can only be used in a call to `SQLGetDiagRec()` to obtain more error information, or to `SQLFreeEnv()`.

SQLAllocEnv

Diagnostics

Table 15. SQLAllocEnv SQLSTATEs

SQLSTATE	Description	Explanation
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.

Restrictions

None.

Example

See “Example” on page 74.

References

- “SQLAllocConnect - Allocate connection handle” on page 73
- “SQLFreeEnv - Free environment handle” on page 191

SQLAllocHandle - Allocate handle

Purpose

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

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

This function is a generalized function for allocating handles that replaces the deprecated ODBC 2.0 functions SQLAllocEnv(), SQLAllocConnect() and SQLAllocStmt().

Syntax

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

Function arguments

Table 16. SQLAllocHandle arguments

Data type	Argument	Use	Description
SQLSMALLINT	<i>HandleType</i>	input	The type of handle to be allocated by SQLAllocHandle(). Must be one of the following values: <ul style="list-style-type: none"> SQL_HANDLE_ENV SQL_HANDLE_DBC SQL_HANDLE_STMT
SQLHANDLE	<i>InputHandle</i>	input	Existing handle to use as a context for the new handle being allocated. If <i>HandleType</i> is: <ul style="list-style-type: none"> SQL_HANDLE_ENV, this is SQL_NULL_HANDLE value(it is ignored). SQL_HANDLE_DBC, this must be the environment handle. SQL_HANDLE_STMT, this must be a connection handle.
SQLHANDLE *	<i>OutputHandlePtr</i>	output	Pointer to a buffer in which to return the newly allocated handle.

Usage

SQLAllocHandle() is used to allocate environment, connection, and statement handles:

- **Allocating an environment handle**

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

On the call to SQLAllocHandle() to request an environment handle, the DB2 ODBC 3.0 driver implicitly sets SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC3. See "ODBC 3.0 driver behavior" on page 503 for more information.

SQLAllocHandle

When the DB2 ODBC 3.0 driver processes the `SQLAllocHandle()` function with a *HandleType* of `SQL_HANDLE_ENV`, it checks the trace keywords in the common section of the DB2 ODBC initialization file. If set, the DB2 ODBC enables tracing if not already started. Tracing ends when the environment handle is freed. See Chapter 7, “Problem diagnosis”, on page 457 and “DB2 ODBC initialization file” on page 52 for more information.

The DB2 ODBC 3.0 driver does not support multiple environments. See “Restrictions” on page 81.

- **Allocating a connection handle**

A connection handle provides access to information such as the valid statement handles on the connection and whether a transaction is currently open. To request a connection handle, an application calls `SQLAllocHandle()` with a *HandleType* of `SQL_HANDLE_DBC`. The *InputHandle* argument is set to the environment handle that was returned by the call to `SQLAllocHandle()` that allocated that handle. DB2 ODBC allocates the connection handle, and passes the value of the associated handle back in **OutputHandlePtr*. The application passes the **OutputHandlePtr* value in all subsequent calls that require a connection handle argument.

- **Allocating a statement handle**

A statement handle provides access to statement information, such as messages, the cursor name, and status information for SQL statement processing. To request a statement handle, an application connects to a data source. The application then calls `SQLAllocHandle()` prior to submitting SQL statements. In this call, *HandleType* should be set to `SQL_HANDLE_STMT` and *InputHandle* should be set to the connection handle that was returned by call to `SQLAllocHandle()` that allocated that handle. DB2 ODBC allocates the statement handle, associates the statement handle with the connection specified, and passes the value of the associated handle back in **OutputHandlePtr*. The application passes the **OutputHandlePtr* value in subsequent calls that require a statement handle argument.

- **Managing handles**

Multiple connection and statement handles can be allocated by an application at the same time. DB2 ODBC 3.0 driver applications can use the same environment, connection, or statement handle on different threads. DB2 ODBC provides thread-safe access for all handles and function calls. Applications can have multiple connections to the same or different data sources at the same time and each connections maintains its own unit of recovery. The application itself might experience unpredictable behavior if the threads it creates do not coordinate their use of DB2 ODBC resources. For example, application behavior might be unpredictable if multiple threads call ODBC functions on the same connection simultaneously. See “Writing multithreaded applications” on page 421 for more information.

If the application calls `SQLAllocHandle()` with **OutputHandlePtr* set to a connection or statement handle that already exists, DB2 ODBC overwrites the information associated with the handle. DB2 ODBC does not check to see whether the handle entered in **OutputHandlePtr* is already in use, nor does it check the previous contents of a handle before overwriting it.

Return codes

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_INVALID_HANDLE`
- `SQL_ERROR`

Environment handle allocation errors: When allocating an environment handle:

- If `SQLAllocHandle()` returns `SQL_ERROR` and `OutputHandlePtr` is set to `SQL_NULL_HENV`, `SQLGetDiagRec()` cannot be called because there is no handle with which to associate additional information.
- If `SQLAllocHandle()` returns `SQL_ERROR` and `OutputHandlePtr` is *not* set to `SQL_NULL_HENV`, the handle is a restricted environment handle. This means that the handle can be used only to call `SQLGetDiagRec()` to obtain more error information or `SQLFreeHandle` to free the restricted environment handle.

Connection or statement handle allocation errors: When allocating a handle other than an environment handle:

- If `SQLAllocHandle()` returns `SQL_ERROR`, it will set `OutputHandlePtr` to `SQL_NULL_HDBC` or `SQL_NULL_HSTMT` depending on the value of `HandleType` (unless the output argument is a null pointer). The application should call `SQLGetDiagRec()` with the environment handle.
- If `SQLAllocHandle()` returns `SQL_SUCCESS_WITH_INFO`, it will set `OutputHandlePtr`. The application can then obtain additional information by calling `SQLGetDiagRec()` with the appropriate `HandleType` and `Handle` set to the value of `SQLAllocHandle()` `HandleType` and `InputHandle`.

Diagnostics

Table 17. `SQLAllocHandle` `SQLSTATE`s

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns <code>SQL_SUCCESS_WITH_INFO</code> .)
08003	Connection is closed.	The <code>HandleType</code> argument was <code>SQL_HANDLE_STMT</code> , but the connection specified by the <code>InputHandle</code> argument was not open. The connection process must be completed successfully (and the connection must be open) for DB2 ODBC to allocate a statement handle.
HY000	General error.	An error occurred for which there was no specific <code>SQLSTATE</code> . The error message returned by <code>SQLGetDiagRec()</code> in the <code>*MessageText</code> buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC was unable to allocate memory for the specified handle.
HY009	Invalid use of a null pointer.	The <code>OutputHandlePtr</code> argument was a null pointer
HY013	Unexpected memory handling error.	The <code>HandleType</code> argument was <code>SQL_HANDLE_DBC</code> or <code>SQL_HANDLE_STMT</code> and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY014	No more handles.	The limit for the number of handles that can be allocated for the type of handle indicated by the <code>HandleType</code> argument has been reached.
HY092	Option type out of range.	The <code>HandleType</code> argument was not: <ul style="list-style-type: none"> • <code>SQL_HANDLE_ENV</code> • <code>SQL_HANDLE_DBC</code> • <code>SQL_HANDLE_STMT</code>

Restrictions

The DB2 ODBC 3.0 driver does not support multiple environments. The DB2 ODBC 3.0 driver behaves like the DB2 ODBC 2.0 driver; there can be only one active environment at any time. Subsequent calls to `SQLAllocHandle()` to allocate another

SQLAllocHandle

environment handle will return the same environment handle and SQL_SUCCESS. The DB2 ODBC driver keeps an internal count of environment requests. SQLFreeHandle() must then be called for each successful SQLAllocHandle() for an environment handle. The last successful SQLFreeHandle() for an environment handle will free the DB2 ODBC 3.0 driver environment. This ensures that the driver environment is not prematurely deallocated under an ODBC application. The DB2 ODBC 2.0 driver and DB2 ODBC 3.0 driver behave consistently in this situation.

Example

Refer to sample program DSN8O3VP in DSN710.SDSNSAMP.

References

- “SQLFreeHandle - Free handle resources” on page 193
- “SQLGetDiagRec - Get multiple field settings of diagnostic record” on page 223
- “SQLSetEnvAttr - Set environment attribute” on page 350
- “SQLSetConnectAttr - Set connection attributes” on page 336
- “SQLSetStmtAttr - Set options related to a statement” on page 360

SQLAllocStmt - Allocate a statement handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

In ODBC 3.0, `SQLAllocHandle()` replaces the ODBC 2.0 function `SQLAllocStmt()`. See `SQLAllocHandle()` for more information.

`SQLAllocStmt()` allocates a new statement handle and associates it with the connection specified by the connection handle. There is no defined limit on the number of statement handles that can be allocated at any one time.

`SQLConnect()` or `SQLDriverConnect()` must be called before calling this function.

This function must be called before `SQLBindParameter()`, `SQLPrepare()`, `SQLExecute()`, `SQLExecDirect()`, or any other function that has a statement handle as one of its input arguments.

Syntax

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

Function arguments

Table 18. *SQLAllocStmt* arguments

Data type	Argument	Use	Description
SQLHDBC	<i>hdbc</i>	input	Connection handle
SQLHSTMT *	<i>phstmt</i>	output	Pointer to statement handle

Usage

DB2 ODBC uses each statement handle to relate all the descriptors, attribute values, result values, cursor information, and status information to the SQL statement processed. Although each SQL statement must have a statement handle, you can reuse the handles for different statements.

A call to this function requires that *hdbc* references an active database connection.

To execute a positioned UPDATE or DELETE, the application must use different statement handles for the SELECT statement and the UPDATE or DELETE statement.

If the input pointer to the statement handle (*phstmt*) already points to a valid statement handle allocated by a previous call to `SQLAllocStmt()`, then the original value is overwritten as a result of this call. This is an application programming error that is not detected by DB2 ODBC.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

SQLAllocStmt

If SQL_ERROR is returned, the *phstmt* argument is set to SQL_NULL_HSTMT. The application should call SQLGetDiagRec() with the same *hdbc* and with the *hstmt* argument set to SQL_NULL_HSTMT.

Diagnostics

Table 19. SQLAllocStmt SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection is closed.	The connection specified by the <i>hdbc</i> argument is not open. The connection must be established successfully (and the connection must be open) for the application to call SQLAllocStmt().
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	<i>phstmt</i> was a null pointer.
S1013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
S1014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.

Restrictions

None.

Example

See “Example” on page 180.

References

- “SQLConnect - Connect to a data source” on page 129
- “SQLDriverConnect - (Expanded) connect to a data source” on page 146
- “SQLFreeStmt - Free (or reset) a statement handle” on page 196
- “SQLGetStmtOption - Returns current setting of a statement option” on page 273
- “SQLSetStmtOption - Set statement option” on page 367

SQLBindCol - Bind a column to an application variable

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLBindCol() is used to associate (bind) columns in a result set to:

- Application variables or arrays of application variables (storage buffers), for all C data types. In this case, data is transferred from the DBMS to the application when SQLFetch() or SQLExtendedFetch() is called. Data conversion can occur as the data is transferred.
- A LOB locator, for LOB columns. In this case a LOB locator, not the data itself, is transferred from the DBMS to the application when SQLFetch() is called. A LOB locator can represent the entire data or a portion of the data.

SQLBindCol() is called once for each column in the result set that the application needs to retrieve.

In general, SQLPrepare(), SQLExecDirect() or one of the schema functions is called before this function, and SQLFetch() or SQLExtendedFetch() is called after. Column attributes might also be needed before calling SQLBindCol(), and can be obtained using SQLDescribeCol() or SQLColAttribute().

Syntax

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

Function arguments

Table 20. SQLBindCol arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle
SQLUSMALLINT	<i>icol</i>	input	Number identifying the column. Columns are numbered sequentially, from left to right, starting at 1.

SQLBindCol

Table 20. SQLBindCol arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	<i>fCType</i>	input	<p>The C data type for column number <i>icol</i> in the result set. The following types are supported:</p> <ul style="list-style-type: none"> • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_WCHAR <p>The supported data types are based on the data source to which you are connected. Specifying SQL_C_DEFAULT causes data to be transferred to its default C data type. See Table 4 on page 31 for more information.</p>
SQLPOINTER	<i>rgbValue</i>	output (deferred)	<p>Pointer to buffer (or an array of buffers if using SQLExtendedFetch()) where DB2 ODBC is to store the column data or the LOB locator when the fetch occurs.</p> <p>If <i>rgbValue</i> is null, the column is unbound.</p>
SQLINTEGER	<i>cbValueMax</i>	input	<p>Size of <i>rgbValue</i> buffer in bytes available to store the column data or the LOB locator.</p> <p>If <i>fCType</i> denotes a binary or character string (either single or double byte) or is SQL_C_DEFAULT, then <i>cbValueMax</i> must be > 0, or an error is returned. Otherwise, this argument is ignored.</p>
SQLINTEGER *	<i>pcbValue</i>	output (deferred)	<p>Pointer to value (or array of values) which indicates the number of bytes DB2 ODBC has available to return in the <i>rgbValue</i> buffer. If <i>fCType</i> is a LOB locator, the size of the locator is returned, not the size of the LOB data.</p> <p>SQLFetch() returns SQL_NULL_DATA in this argument if the data value of the column is null.</p> <p>This pointer value must be unique for each bound column, or NULL.</p> <p>SQL_NO_LENGTH can also be returned. See the 'Usage' section below for more information.</p>

Note:

- For this function, pointers *rgbValue* and *pcbValue* are deferred outputs, meaning that the storage locations they point to do not get updated until a result set row is fetched. As a result, the locations referenced by these pointers must remain valid until SQLFetch() or SQLExtendedFetch() is called. For example, if SQLBindCol() is called within a local function, SQLFetch() must be called from within the same scope of the function or the *rgbValue* buffer must be allocated as static or global.

- DB2 ODBC performs better for all variable length data types if *rgbValue* is placed consecutively in memory after *pcbValue*. See the 'Usage' section for more details.

Usage

The application calls `SQLBindCol()` once for each column in the result set for which it wishes to retrieve either the data, or optionally in the case of LOB columns, a LOB locator. Result sets are generated either by calling `SQLPrepare()`, `SQLExecDirect()`, `SQLGetTypeInfo()`, or one of the catalog functions. When `SQLFetch()` is called, the data in each of these *bound* columns is placed into the assigned location (given by the pointers *rgbValue* and *cbValue*). If *fCType* is a LOB locator, a locator value is returned (not the LOB data). The LOB locator references the entire data value in the LOB column.

`SQLExtendedFetch()` can be used in place of `SQLFetch()` to retrieve multiple rows from the result set into an array. In this case, *rgbValue* references an array. For more information, see “Retrieving a result set into an array” on page 406 and “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169. Use of `SQLExtendedFetch()` and `SQLFetch()` cannot be mixed for the same result set.

Columns are identified by a number, assigned sequentially from left to right, starting at 1. The number of columns in the result set can be determined by calling `SQLNumResultCols()` or by calling `SQLColAttribute()` with the *fDescType* argument set to `SQL_COLUMN_COUNT`.

The application can query the attributes (such as data type and length) of the column by first calling `SQLDescribeCol()` or `SQLColAttribute()`. (As an alternative, see “Programming hints and tips” on page 452 for information about using `SQLSetColAttributes()` when the application has prior knowledge of the format of the result set.) This information can then be used to allocate a storage location of the correct data type and length to indicate data conversion to another data type, or in the case of LOB data types, optionally return a locator. See “Data types and data conversion” on page 30 for more information on default types and supported conversions.

An application can choose not to bind every column, or even not to bind any columns. Data in any of the columns can also be retrieved using `SQLGetData()` after the bound columns have been fetched for the current row. Generally, `SQLBindCol()` is more efficient than `SQLGetData()`. For a discussion of when to use one function over the other, refer to “Programming hints and tips” on page 452.

In subsequent fetches, the application can change the binding of these columns or bind previously unbound columns by calling `SQLBindCol()`. The new binding does not apply to data already fetched, it is used on the next fetch. To unbind a single column, call `SQLBindCol()` with the *rgbValue* pointer set to `NULL`. To unbind all the columns, the application should call `SQLFreeStmt()` with the *fOption* input set to `SQL_UNBIND`.

The application must ensure enough storage is allocated for the data to be retrieved. If the buffer is to contain variable length data, the application must allocate as much storage as the maximum length of the bound column requires; otherwise, the data might be truncated. If the buffer is to contain fixed length data, DB2 ODBC assumes the size of the buffer is the length of the C data type. If data conversion is specified, the required size might be affected, see “Data types and data conversion” on page 30 for more information.

If string truncation does occur, `SQL_SUCCESS_WITH_INFO` is returned and `pcbValue` is set to the actual size of `rgbValue` available for return to the application.

Truncation is also affected by the `SQL_MAX_LENGTH` statement option (used to limit the amount of data returned to the application). The application can specify not to report truncation by calling `SQLSetStmtAttr()` with `SQL_MAX_LENGTH` and a value for the maximum length to return for all variable length columns, and by allocating an `rgbValue` buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, `SQL_SUCCESS` is returned when the value is fetched and the maximum length, not the actual length, is returned in `pcbValue`.

If the column to be bound is an `SQL_GRAPHIC`, `SQL_VARGRAPHIC` or `SQL_LONGVARGRAPHIC` type, then `fCType` can be set to `SQL_C_DBCHAR` or `SQL_C_CHAR`. If `fCType` is `SQL_C_DBCHAR`, the data fetched into the `rgbValue` buffer is null-terminated by a double byte null-terminator. If `fCType` is `SQL_C_CHAR`, then the data is not null-terminated. In both cases, the length of the `rgbValue` buffer (`cbValueMax`) is in units of bytes and should therefore be a multiple of 2.

When binding any variable length column, DB2 ODBC can write `pcbValue` and `rgbValue` in one operation if they are allocated contiguously. For example:

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

Note: `SQL_NO_TOTAL` is returned in `pcbValue` if:

- The SQL type is a variable length type, and
- `pcbValue` and `rgbValue` are contiguous, and
- The column type is NOT NULLABLE, and
- String truncation occurred.

If the column to be bound is an `SQL_ROWID` type, then `fCType` can be set to `SQL_C_CHAR` or `SQL_C_DEFAULT`. The data fetched into the `rgbValue` buffer is null-terminated. The maximum length of a ROWID column in the DBMS is 40 bytes, therefore, the application must allocate an `rgbValue` buffer of at least 40 bytes for the data to be retrieved without truncation.

To retrieve UCS-2 data, set `fCType` to `SQL_C_WCHAR`. The retrieved data is terminated by a double-byte null terminator.

LOB locators are generally treated like any other data type, but there are some important differences:

- Locators are generated at the server when a row is fetched and a LOB locator C data type is specified on `SQLBindCol()`, or when `SQLGetSubString()` is called to define a locator on a portion of another LOB. Only the locator is transferred to the application.
- The value of the locator is only valid within the current transaction. You cannot store a locator value and use it beyond the current transaction, even if the cursor used to fetch the LOB locator has the WITH HOLD attribute.
- A locator can be freed before the end of the transaction using the `FREE LOCATOR` statement.

- When a locator is received, the application can use `SQLGetSubString()` to either receive a portion of the LOB value, or to generate another locator representing the substring. The locator value can also be used as input for a parameter marker (using `SQLBindParameter()`).

A LOB locator is not a pointer to a database position; rather it is a reference to a LOB value, a snapshot of that LOB value. There is no association between the current position of the cursor and the row from which the LOB value is extracted. Therefore, even after the cursor moves to a different row, the LOB locator {and thus the value that it represents) can still be referenced.

- `SQLGetPosition()` and `SQLGetLength()` can be used with `SQLGetSubString()` to define the substring.

A given LOB column in the result set can be bound to one of the following:

- Storage buffer, for holding the entire LOB data value
- LOB locator

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

Return codes

- `SQL_SUCCESS`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`

Diagnostics

Table 21. *SQLBindCol* SQLSTATEs

SQLSTATE	Description	Explanation
07009	Invalid descriptor index.	The value specified for <i>icol</i> was less than 0 or the value specified for the argument <i>icol</i> was greater than the number of columns in the result set.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	The argument <i>TargetType</i> was not a valid data type or <code>SQL_C_DEFAULT</code> .
HY010	Function sequence error.	The function is called while in a data-at-execute (<code>SQLParamData()</code> , <code>SQLPutData()</code>) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument <i>cbValueMax</i> is less than 0.
HYC00	Driver not capable.	The value specified for the argument <i>fctype</i> was not supported by DB2 ODBC.

Note: Additional diagnostic messages relating to the bound columns might be reported at fetch time.

Restrictions

None.

SQLBindCol

Example

See “Example” on page 180.

References

- “SQLFetch - Fetch next row” on page 176
- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169

SQLBindParameter - Binds a parameter marker to a buffer or LOB locator

Purpose

Specification:	ODBC 2.0		
-----------------------	-----------------	--	--

SQLBindParameter() associates (bind) parameter markers in an SQL statement to either:

- Application variables or arrays of application variables (storage buffers), for all C data types. In this case, data is transferred from the application to the DBMS when SQLExecute() or SQLExecDirect() is called. Data conversion can occur as the data is transferred.
- A LOB locator, for SQL LOB data types. In this case, the application transfers a LOB locator value (not the LOB data itself) to the server when the SQL statement is executed.

This function must also be used to bind application storage to a parameter of a stored procedure CALL statement where the parameter can be input, output or both. This function is essentially an extension of SQLSetParam().

Syntax

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

Function arguments

Table 22. SQLBindParameter arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle
SQLUSMALLINT	ipar	input	Parameter marker number, ordered sequentially left to right, starting at 1.

SQLBindParameter

Table 22. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	fParamType	input	<p>The type of parameter. The supported types are:</p> <ul style="list-style-type: none"> SQL_PARAM_INPUT: The parameter marker is associated with an SQL statement that is not a stored procedure CALL; or, it marks an input parameter of the CALLED stored procedure. When the statement is executed, actual data value for the parameter is sent to the server: the <i>rgbValue</i> buffer must contain valid input data values; the <i>pcbValue</i> buffer must contain the corresponding length value or SQL_NTS, SQL_NULL_DATA, or (if the value should be sent using SQLParamData() and SQLPutData()) SQL_DATA_AT_EXEC. SQL_PARAM_INPUT_OUTPUT: The parameter marker is associated with an input/output parameter of the CALLED stored procedure. When the statement is executed, actual data value for the parameter is sent to the server: the <i>rgbValue</i> buffer must contain valid input data values; the <i>pcbValue</i> buffer must contain the corresponding length value or SQL_NTS, SQL_NULL_DATA, or (if the value should be sent using SQLParamData() and SQLPutData()) SQL_DATA_AT_EXEC. SQL_PARAM_OUTPUT: The parameter marker is associated with an output parameter of the CALLED stored procedure or the return value of the stored procedure. After the statement is executed, data for the output parameter is returned to the application buffer specified by <i>rgbValue</i> and <i>pcbValue</i>, unless both are NULL pointers, in which case the output data is discarded.
SQLSMALLINT	fCType	input	<p>C data type of the parameter. The following types are supported:</p> <ul style="list-style-type: none"> SQL_C_BINARY SQL_C_BIT SQL_C_BLOB_LOCATOR SQL_C_CHAR SQL_C_CLOB_LOCATOR SQL_C_DBCHAR SQL_C_DBCLOB_LOCATOR SQL_C_DOUBLE SQL_C_FLOAT SQL_C_LONG SQL_C_SHORT SQL_C_TYPE_DATE SQL_C_TYPE_TIME SQL_C_TYPE_TIMESTAMP SQL_C_TINYINT SQL_C_WCHAR <p>Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type to the type indicated in <i>fSqlType</i>.</p>

Table 22. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	fSqlType	input	<p>SQL data type of the parameter. The supported types are:</p> <ul style="list-style-type: none"> • SQL_BINARY • SQL_BLOB • SQL_BLOB_LOCATOR • SQL_CHAR • SQL_CLOB • SQL_CLOB_LOCATOR • SQL_DBCLOB • SQL_DBCLOB_LOCATOR • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_ROWID • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC <p>Note: 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.</p>
SQLINTEGER	cbColDef	input	<p>Precision of the corresponding parameter marker. If <i>fSqlType</i> denotes:</p> <ul style="list-style-type: none"> • A binary or single byte character string (for example, SQL_CHAR, SQL_BINARY), this is the maximum length in bytes for this parameter marker. • A double byte character string (for example, SQL_GRAPHIC), this is the maximum length in double-byte characters for this parameter. • SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision. • SQL_ROWID, this must be set to 40, the maximum length in bytes for this data type. Otherwise, an error is returned. • Otherwise, this argument is ignored.
SQLSMALLINT	ibScale	input	<p>Scale of the corresponding parameter if <i>fSqlType</i> is SQL_DECIMAL or SQL_NUMERIC. If <i>fSqlType</i> is SQL_TYPE_TIMESTAMP, this is the number of digits to the right of the decimal point in the character representation of a timestamp (for example, the scale of yyyy-mm-dd hh:mm:ss.fff is 3).</p> <p>Other than for the <i>fSqlType</i> values mentioned here, <i>ibScale</i> is ignored.</p>

SQLBindParameter

Table 22. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLPOINTER	rgbValue	input (deferred) and/or output (deferred)	<ul style="list-style-type: none"> On input (<i>fParamType</i> set to SQL_PARAM_INPUT, or SQL_PARAM_INPUT_OUTPUT): <p>At execution time, if <i>pcbValue</i> does not contain SQL_NULL_DATA or SQL_DATA_AT_EXEC, then <i>rgbValue</i> points to a buffer that contains the actual data for the parameter.</p> <p>If <i>pcbValue</i> contains SQL_DATA_AT_EXEC, then <i>rgbValue</i> 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.</p> <p>If SQLParamOptions() is called to specify multiple values for the parameter, then <i>rgbValue</i> is a pointer to an input buffer array of <i>cbValueMax</i> bytes.</p> On output (<i>fParamType</i>) set to SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT): <p><i>rgbValue</i> points to the buffer where the output parameter value of the stored procedure is stored.</p> <p>If <i>fParamType</i> is set to SQL_PARAM_OUTPUT, and both <i>rgbValue</i> and <i>pcbValue</i> are NULL pointers, then the output parameter value or the return value from the stored procedure call is discarded.</p>
SQLINTEGER	cbValueMax	input	<p>For character and binary data, <i>cbValueMax</i> specifies the length of the <i>rgbValue</i> buffer (if treated as a single element) or the length of each element in the <i>rgbValue</i> array (if the application calls SQLParamOptions() to specify multiple values for each parameter). For non-character and non-binary data, this argument is ignored -- the length of the <i>rgbValue</i> buffer (if it is a single element) or the length of each element in the <i>rgbValue</i> array (if SQLParamOptions() is used to specify an array of values for each parameter) is assumed to be the length associated with the C data type.</p> <p>For output parameters, <i>cbValueMax</i> is used to determine whether to truncate character or binary output data in the following manner:</p> <ul style="list-style-type: none"> For character data, if the number of bytes available to return is greater than or equal to <i>cbValueMax</i>, the data in <i>rgbValue</i> is truncated to <i>cbValueMax-1</i> bytes and is null-terminated (unless null-termination has been turned off). For binary data, if the number of bytes available to return is greater than <i>cbValueMax</i>, the data in <i>rgbValue</i> is truncated to <i>cbValueMax</i> bytes.

Table 22. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER *	pcbValue	input (deferred) and/or output (deferred)	<p>- If this is an input or input/output parameter:</p> <p>This is the pointer to the location which contains (when the statement is executed) the length of the parameter marker value stored at <i>rgbValue</i>.</p> <p>To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA.</p> <p>If <i>fCType</i> is SQL_C_CHAR or SQL_C_WCHAR, this storage location must contain either the exact length of the data stored at <i>rgbValue</i>, or SQL_NTS if the contents at <i>rgbValue</i> are null-terminated.</p> <p>If <i>fCType</i> indicates character data (explicitly, or implicitly using SQL_C_DEFAULT), and this pointer is set to NULL, it is assumed that the application always provides a null-terminated string in <i>rgbValue</i>. This also implies that this parameter marker never has a null value.</p> <p>If <i>fSqlType</i> denotes a graphic data type and the <i>fCType</i> is SQL_C_CHAR, the pointer to <i>pcbValue</i> can never be NULL and the contents of <i>pcbValue</i> can never hold SQL_NTS. In general for graphic data types, this length should be the number of octets that the double byte data occupies; therefore, the length should always be a multiple of 2. In fact, if the length is odd, then an error occurs when the statement is executed.</p> <p>When SQLExecute() or SQLExecDirect() is called, and <i>pcbValue</i> 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 <i>data-at-execution</i> parameter.</p> <p>If SQLParamOptions() is used to specify multiple values for each parameter, <i>pcbValue</i> points to an array of SQLINTEGER values where each of the elements can be the number of bytes in the corresponding <i>rgbValue</i> element (excluding the null-terminator), or SQL_NULL_DATA.</p> <p>- If this is an output parameter (<i>fParamType</i> is set to SQL_PARAM_OUTPUT):</p> <p>This must be an output parameter or return value of a stored procedure CALL and points to one of the following, after the execution of the stored procedure:</p> <ul style="list-style-type: none"> • number of bytes available to return in <i>rgbValue</i>, excluding the null-termination character. • SQL_NULL_DATA • SQL_NO_TOTAL if the number of bytes available to return cannot be determined.

Usage

A parameter marker is represented by a "?" character in an SQL statement and is used to indicate a position in the statement where an application supplied value is to be substituted when the statement is executed. This value can be obtained from:

SQLBindParameter

- An application variable. `SQLBindParameter()` (or `SQLSetParam()`) is used to bind the application storage area to the parameter marker.
- A LOB value from the database server (by specifying a LOB locator).
`SQLBindParameter()` (or `SQLSetParam()`) is used to bind a LOB locator to the parameter marker. The LOB value itself is supplied by the database server, so only the LOB locator is transferred between the database server and the application.
An application can use a locator with `SQLGetSubstring()`, `SQLGetPosition()`, or `SQLGetLength()`. `SQLGetSubstring()` can either return another locator or the data itself. All locators remain valid until the end of the transaction in which they are created (even when the cursor moves to another row), or until it is freed using the `FREE LOCATOR` statement.

The application must bind a variable to each parameter marker in the SQL statement before executing the SQL statement. For this function, *rgbValue* and *pcbValue* are deferred arguments, the storage locations must be valid and contain input data values when the statement is executed. This means either keeping the `SQLExecDirect()` or `SQLExecute()` call in the same procedure scope as the `SQLBindParameter()` calls, or, these storage locations must be dynamically allocated or declared statically or globally.

`SQLSetParam()` can be called before `SQLPrepare()` if the columns in the result set are known; otherwise, the attributes of the result set can be obtained after the statement is prepared.

Parameter markers are referenced by number (*icol*) and are numbered sequentially from left to right, starting at 1.

All parameters bound by this function remain in effect until:

- `SQLFreeHandle()` is called with *HandleType* set to `SQL_HANDLE_STMT`, or
- `SQLFreeStmt()` is called with the `SQL_RESET_PARAMS` option, or
- `SQLBindParameter()` is called again for the same parameter *ipar* number.

After the SQL statement is executed, and the results processed, the application might wish to reuse the statement handle to execute a different SQL statement. If the parameter marker specifications are different (number of parameters, length or type), `SQLFreeStmt()` should be called 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.

An application can pass the value for a parameter either in the *rgbValue* buffer or with one or more calls to `SQLPutData()`. In the latter case, 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.

Since 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.

SQLBindParameter() essentially extends the capability of the SQLSetParam() function by providing a method of:

- Specifying whether a parameter is input, input / output, or output, necessary for proper handling of parameters for stored procedures.
- Specifying an array of input parameter values when SQLParamOptions() is used in conjunction with SQLBindParameter(). SQLSetParam() can still be used to bind single element application variables to parameter markers that are not part of a stored procedure CALL statement.

The *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.

Note:

- If an application cannot determine the type of a parameter in a procedure call, set *fParamType* 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 *pcbValue* buffer to SQL_NULL_DATA.
- If an application marks a parameter as SQL_PARAM_OUTPUT, data for the parameter is returned to the application after the CALL statement 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.
- For this function, *rgbValue* and *pcbValue* are deferred arguments. In the case where *fParamType* is set to SQL_PARAM_INPUT or SQL_PARAM_INPUT_OUTPUT, the storage locations must be valid and contain input data values when the statement is executed. This means either keeping the SQLExecDirect() or SQLExecute() call in the same procedure scope as the SQLBindParameter() calls, or, these storage locations must be dynamically allocated or statically or globally declared.
Similarly, if *fParamType* is set to SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT, the *rgbValue* and *pcbValue* buffer locations must remain valid until the CALL statement is executed.
- If *fSqlType* is SQL_ROWID, the *cbColDef* value must be set to 40, the maximum length in bytes for a ROWID data type. If *cbColDef* is not set to 40, the application receives SQLSTATE=22001 when *cbColDef* is less than 40 and SQLSTATE=HY104 when *cbColDef* is greater than 40.

For character and binary C data, the *cbValueMax* argument specifies the length of the *rgbValue* buffer if it is a single element; or, if the application calls SQLParamOptions() to specify multiple values for each parameter, *cbValueMax* is the length of *each* element in the *rgbValue* array, INCLUDING the null-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.

An application can pass the value for a parameter either in the *rgbValue* buffer or with one or more calls to SQLPutData(). In the latter case, these parameters are

SQLBindParameter

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.

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;
```

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 23. SQLBindParameter SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The conversion from the data value identified by the fCType argument to the data type identified by the fSqlType argument is not a meaningful conversion. (For example, conversion from SQL_C_TYPE_DATE to SQL_DOUBLE.)
07009	Invalid descriptor index.	The value specified for ColumnNumber was less than 0 or the value specified for the argument ColumnNumber was greater than the number of columns in the result set.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	The argument TargetType was not a valid data type or SQL_C_DEFAULT.
HY004	Invalid SQL data type.	The value specified for the argument fSqlType is not a valid SQL data type.
HY009	Invalid use of a null pointer.	The argument OutputHandlePtr is a null pointer.
HY010	Function sequence error.	Function is called after SQLExecute() or SQLExecDirect() returned SQL_NEED_DATA, but data have not been sent for all data-at-execution parameters.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument BufferLength is less than 0.

Table 23. SQLBindParameter SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY104	Invalid precision or scale value.	The value specified for <i>fSqlType</i> was either SQL_DECIMAL or SQL_NUMERIC and the value specified for <i>cbParamDef</i> was less than 1. The value specified for <i>fCType</i> is SQL_C_TYPE_TIMESTAMP; the value for <i>fSqlType</i> is either SQL_CHAR or SQL_VARCHAR; and the value for <i>ibScale</i> is less than 0 or greater than 6.
HY105	Invalid parameter type.	<i>fParamType</i> is not one of SQL_PARAM_INPUT, SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT.
HYC00	Driver not capable.	DB2 ODBC or data source does not support the conversion specified by the combination of the value specified for the argument <i>fCType</i> and the value specified for the argument <i>fSqlType</i> . The value specified for the argument <i>fSqlType</i> is not supported by either DB2 ODBC or the data source.

Restrictions

In ODBC 2.0, this function has replaced SQLSetParam().

A new value for *pcbValue*, 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. Since DB2 stored procedure arguments do not have the concept of default values, specification of this value for *pcbValue* argument results in an error when the CALL statement is executed since 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 of the entire data that is sent for character or binary C data using the subsequent SQLPutData() calls. Since the DB2 ODBC driver does not need this information, the macro is not needed. An ODBC application calls SQLGetInfo() with the SQL_NEED_LONG_DATA_LEN option to check if the driver needs this information. The DATABASE 2 ODBC driver returns 'N' to indicate that this information is not needed by SQLPutData().

Example

The example shown below binds a variety of data types to a set of parameters. For an additional example see Appendix F, "Example code", on page 507.

SQLBindParameter

```
/* ... */
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"
};

SQLINTEGER   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 %ld Rows\n", pirow);
/* ... */
```

References

- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLExecute - Execute a statement” on page 166

- “SQLParamData - Get next parameter for which a data value is needed” on page 296
- “SQLParamOptions - Specify an input array for a parameter” on page 298
- “SQLPutData - Passing data value for a parameter” on page 327

SQLCancel - Cancel statement

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLCancel() can be used to prematurely terminate the *data-at-execution* sequence described in “Sending/retrieving long data in pieces” on page 401.

Syntax

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

Function arguments

Table 24. SQLCancel arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle

Usage

After SQLExecDirect() or SQLExecute() returns SQL_NEED_DATA to solicit for values for data-at-execution parameters, SQLCancel() can be used to cancel the data-at-execution sequence described in “Sending/retrieving long data in pieces” on page 401. SQLCancel() can be called any time before the final SQLParamData() in the sequence. After the cancellation of this sequence, the application can call SQLExecute() or SQLExecDirect() to re-initiate the data-at-execution sequence.

If an application calls SQLCancel() on an *hstmt* not associated with a data-at-execution sequence, SQLCancel() has the same effect as SQLFreeHandle() with the *HandleType* set to SQL_HANDLE_STMT. Applications should not call SQLCancel() to close a cursor; but rather SQLFreeStmt() should be used.

Return codes

- SQL_SUCCESS
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

Table 25. SQLCancel SQLSTATES

SQLSTATE	Description	Explanation
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.

Restrictions

DB2 ODBC does not support asynchronous statement execution.

Example

See “Example” on page 329.

References

- “SQLPutData - Passing data value for a parameter” on page 327
- “SQLParamData - Get next parameter for which a data value is needed” on page 296

SQLCloseCursor - Close cursor and discard pending results

Purpose

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

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

Syntax

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

Function arguments

Table 26. 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 24000 (invalid cursor state) if no cursor is open. Calling SQLCloseCursor() is equivalent to calling the ODBC 2.0 function SQLFreeStmt() with the SQL_CLOSE option. An exception is that SQLFreeStmt() with SQL_CLOSE has no effect on the application if no cursor is open on the statement, while SQLCloseCursor() returns SQLSTATE 24000 (invalid cursor state).

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

Table 27. SQLCloseCursor SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state.	No cursor was open on the <i>StatementHandle</i> .
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by SQLGetDiagRec() in the <i>*MessageText</i> buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC was unable to allocate memory required to support execution or completion of the function.

Table 27. SQLCloseCursor SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() was called for the <i>StatementHandle</i> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY013	Unexpected memory handling error.	DB2 ODBC was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

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

References

- “SQLSetStmtAttr - Set options related to a statement” on page 360
- “SQLSetConnectAttr - Set connection attributes” on page 336
- “SQLGetConnectAttr - Get current attribute setting” on page 199

SQLColAttribute - Get column attributes

Purpose

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

SQLColAttribute() returns descriptor information for a column in a result set. Descriptor information is returned as a character string, a 32-bit descriptor-dependent value, or an integer value.

Syntax

```
SQLRETURN SQLColAttribute (SQLHSTMT      StatementHandle,
                          SQLSMALLINT    ColumnNumber,
                          SQLSMALLINT    FieldIdentifier,
                          SQLPOINTER     CharacterAttributePtr,
                          SQLSMALLINT    BufferLength,
                          SQLSMALLINT    *StringLengthPtr,
                          SQLPOINTER     NumericAttributePtr);
```

Function arguments

Table 28. SQLColAttribute

Data type	Argument	Use	Description
SQLHSTMT	<i>StatementHandle</i>	input	Statement handle.
SQLSMALLINT	<i>ColumnNumber</i>	input	Column number to be described. Columns are numbered sequentially from left to right, starting at 1. Column zero might not be defined. The DB2 ODBC 3.0 driver does not support bookmarks. See "Restrictions" on page 113.
SQLSMALLINT	<i>FieldIdentifier</i>	input	The field in row <i>ColumnNumber</i> that will be returned. See Table 29 on page 107.
SQLPOINTER	<i>CharacterAttributePtr</i>	output	Pointer to a buffer in which to return the value in the <i>FieldIdentifier</i> field of the <i>ColumnNumber</i> row if the field is a character string. Otherwise, the field is not used.
SQLSMALLINT	<i>BufferLength</i>	input	The length of the <i>*CharacterAttributePtr</i> buffer, if the field is a character string. Otherwise, this field is ignored.
SQLSMALLINT *	<i>StringLengthPtr</i>	output	Pointer to a buffer in which to return the total number of bytes (excluding the null termination byte for character data) available to return in <i>*CharacterAttributePtr</i> . For character data, if the number of bytes available to return is greater than or equal to <i>BufferLength</i> , the descriptor information in <i>*CharacterAttributePtr</i> is truncated to <i>BufferLength</i> minus the length of a null termination character and is null-terminated by DB2 ODBC. For all other types of data, the value of <i>BufferLength</i> is ignored and DB2 ODBC assumes the size of <i>*CharacterAttributePtr</i> is 32 bits.
SQLPOINTER	<i>NumericAttributePtr</i>	output	Pointer to an integer buffer in which to return the value in the <i>FieldIdentifier</i> field of the <i>ColumnNumber</i> row, if the field is a numeric descriptor type, such as SQL_DESC_COLUMN_LENGTH. Otherwise, the field is unused.

Usage

SQLColAttribute() returns information either in **NumericAttributePtr* or in **CharacterAttributePtr*. Integer information is returned in **NumericAttributePtr* as a 32-bit, signed value; all other formats of information are returned in **CharacterAttributePtr*. When information is returned in **NumericAttributePtr*, DB2 ODBC ignores *CharacterAttributePtr*, *BufferLength* and *StringLengthPtr*. When information is returned in **CharacterAttributePtr*, DB2 ODBC ignores *NumericAttributePtr*.

ODBC 3.0 SQLColAttribute() replaces the deprecated ODBC 1.0 function SQLDescribeCol() and ODBC 2.0 function SQLColAttribute(). You can continue calling SQLDescribeCol() and SQLColAttribute(). SQLColAttribute() allows access to the more extensive set of descriptor information available in ANSI SQL-92 and DBMS vendor extensions.

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

Table 29 lists the descriptor types returned by ODBC 3.0 SQLColAttribute() and notes (in parentheses) the ODBC 2.0 SQLColAttribute() attribute values replaced or renamed.

Table 29. SQLColAttribute FieldIdentifiers

FieldIdentifier	Information returned in arguments	Description
SQL_DESC_AUTO_UNIQUE_VALUE (SQL_COLUMN_AUTO_INCREMENT) ¹	Numeric AttributePtr	Indicates if the column data type is an autoincrementing data type. SQL_FALSE is returned in <i>NumericAttributePtr</i> for all DB2 SQL data types.
SQL_DESC_BASE_COLUMN_NAME ¹	CharacterAttributePtr	The base column name for the set column. If a base column name does not exist (for example, columns that are expressions), this variable contains an empty string. SQL_DESC_BASE_COLUMN_NAME record
SQL_DESC_BASE_TABLE_NAME ¹	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 <i>NumericAttributePtr</i> , depending on the data type. Case sensitivity does not apply to graphic data types. SQL_FALSE is returned for non-character data types.
SQL_DESC_CATALOG_NAME (SQL_COLUMN_CATALOG_NAME) ¹ (SQL_COLUMN_QUALIFIER_NAME) ¹	CharacterAttributePtr	The catalog of the table that contains the column is returned in <i>CharacterAttributePtr</i> . An empty string is returned because DB2 ODBC supports two part naming for a table.
SQL_DESC_CONCISE_TYPE ¹	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 is returned in <i>NumericAttributePtr</i> .

SQLColAttribute

Table 29. SQLColAttribute FieldIdentifiers (continued)

FieldIdentifier	Information returned in arguments	Description
SQL_DESC_DISPLAY_SIZE (SQL_COLUMN_DISPLAY_SIZE) ¹	<i>NumericAttributePtr</i>	The maximum number of bytes needed to display the data in character form is returned in <i>NumericAttributePtr</i> . See Appendix D, "Data conversion", on page 485 for details about the display size of each of the column data types.
SQL_DESC_DISTINCT_TYPE (SQL_COLUMN_DISTINCT_TYPE) ¹	<i>CharacterAttributePtr</i>	The distinct type name of the column is returned in <i>CharacterAttributePtr</i> . If the column is a built-in SQL type and not a distinct type, an empty string is returned. This is an IBM-defined extension to the list of descriptor attributes that ODBC defines.
SQL_DESC_FIXED_PREC_SCALE (SQL_COLUMN_MONEY) ¹	<i>NumericAttributePtr</i>	SQL_TRUE if the column has a fixed precision and non-zero scale that are data-source-specific. SQL_FALSE if the column does not have a fixed precision and non-zero scale that are data-source-specific. SQL_FALSE is returned in <i>NumericAttributePtr</i> for all DB2 SQL data types.
SQL_DESC_LABEL (SQL_COLUMN_LABEL) ¹	<i>CharacterAttributePtr</i>	The column label is returned in <i>CharacterAttributePtr</i> . If the column does not have a label, the column name or the column expression is returned. If the column is not labeled and named, an empty string is returned.
SQL_DESC_LENGTH ¹	<i>NumericAttributePtr</i>	A numeric value that is either the maximum or actual character length of a character string or binary data type. It is the maximum character length for a fixed-length data type, or the actual character length for a variable-length data type. Its value always excludes the null termination byte that ends the character string.
SQL_DESC_LITERAL_PREFIX ¹	<i>CharacterAttributePtr</i>	This VARCHAR(128) record field 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 ¹	<i>CharacterAttributePtr</i>	This VARCHAR(128) record field contains the character or characters that DB2 ODBC recognizes as a suffix for a literal of this data type. This field contains an empty string if a literal prefix is not applicable to this data type.

Table 29. SQLColAttribute FieldIdentifiers (continued)

FieldIdentifier	Information returned in arguments	Description
SQL_DESC_LOCAL_TYPE_NAME ¹	CharacterAttributePtr	This VARCHAR(128) record field contains any localized (native language) name for the data type that might be different from the regular name of the data type. If there is no localized name, an empty string is returned. This field is for display purposes only. The character set of the string is locale-dependent; it is typically the default character set of the server.
SQL_DESC_NAME (SQL_COLUMN_NAME) ¹	CharacterAttributePtr	The name of the column <i>ColumnNumber</i> is returned in <i>CharacterAttributePtr</i> . If the column is an expression, the column number is returned. In either case, SQL_DESC_UNNAMED is set to SQL_NAMED. If there is no column name or column 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 identified by <i>ColumnNumber</i> can contain nulls, SQL_NULLABLE is returned in <i>NumericAttributePtr</i> . If the column is constrained not to accept nulls, SQL_NO_NULLS is returned in <i>NumericAttributePtr</i> .
SQL_DESC_NUM_PREX_RADIX ¹	NumericAttributePtr	<ul style="list-style-type: none"> If the datatype in the SQL_DESC_TYPE field is an approximate data type, this SQLINTEGER field contains a value of 2, because the SQL_DESC_PRECISION field contains the number of bits. If the datatype in the SQL_DESC_TYPE field is an exact numeric data type, this field contains a value of 10, because the SQL_DESC_PRECISION field contains the number of decimal digits. This field is set to 0 for all non-numeric data types.

SQLColAttribute

Table 29. SQLColAttribute FieldIdentifiers (continued)

FieldIdentifier	Information returned in arguments	Description
SQL_DESC_OCTET_LENGTH (SQL_COLUMN_LENGTH) ¹	<i>NumericAttributePtr</i>	<p>The number of bytes of data associated with the column is returned in <i>NumericAttributePtr</i>. This is the length in bytes of data transferred on the fetch or SQLGetData() for this column if SQL_C_DEFAULT is specified as the C data type. See Appendix D, "Data conversion", on page 485 for details about the length of each of the SQL data types.</p> <p>If the column identified in <i>ColumnNumber</i> is a fixed length character or binary string, (for example, SQL_CHAR or SQL_BINARY), the actual length is returned.</p> <p>If the column identified in <i>ColumnNumber</i> is a variable length character or binary string, (for example, SQL_VARCHAR or SQL_BLOB), the maximum length is returned.</p>
SQL_DESC_PRECISION (SQL_COLUMN_PRECISION) ¹	<i>NumericAttributePtr</i>	<p>The precision in units of digits is returned in <i>NumericAttributePtr</i> if the column is:</p> <ul style="list-style-type: none"> • SQL_DECIMAL • SQL_NUMERIC • SQL_DOUBLE • SQL_FLOAT • SQL_INTEGER • SQL_REAL • SQL_SMALLINT <p>If the column is a character SQL data type, the precision returned in <i>NumericAttributePtr</i>, indicates the maximum number of characters the column can hold.</p> <p>If the column is a graphic SQL data type, the precision returned in <i>NumericAttributePtr</i>, indicates the maximum number of double-byte characters the column can hold. See Appendix D, "Data conversion", on page 485 for information about the precision of each of the SQL data types.</p>
SQL_DESC_SCALE (SQL_COLUMN_SCALE) ¹	<i>NumericAttributePtr</i>	<p>The scale attribute of the column is returned. See Appendix D, "Data conversion", on page 485 for information about the precision of each of the SQL data types.</p>
SQL_DESC_SCHEMA_NAME (SQL_COLUMN_OWNER_NAME) ¹	<i>CharacterAttributePtr</i>	<p>The schema of the table that contains the column is returned in <i>CharacterAttributePtr</i>. An empty string is returned; DB2 is not able to determine this attribute.</p>

Table 29. SQLColAttribute FieldIdentifiers (continued)

FieldIdentifier	Information returned in arguments	Description
SQL_DESC_SEARCHABLE (SQL_COLUMN_SEARCHABLE) ¹	<i>NumericAttributePtr</i>	Indicates if the column data type is searchable: <ul style="list-style-type: none"> • SQL_PRED_NONE (SQL_UNSEARCHABLE in ODBC 2.0) if the column cannot be used in a WHERE clause. • SQL_PRED_CHAR (SQL_LIKE_ONLY in ODBC 2.0) if the column can be used in a WHERE clause only with the LIKE predicate. • SQL_PRED_BASIC (SQL_ALL_EXCEPT_LIKE in ODBC 2.0) if the column can be used in a WHERE clause with all comparison operators except LIKE. • SQL_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.
SQL_DESC_TABLE_NAME (SQL_COLUMN_TABLE_NAME) ¹	<i>CharacterAttributePtr</i>	The name of the table that contains the column is returned in <i>CharacterAttributePtr</i> . An empty string is returned; DB2 ODBC cannot determine this attribute.
SQL_DESC_TYPE (SQL_COLUMN_TYPE) ¹	<i>NumericAttributePtr</i>	The SQL data type of the column identified in <i>ColumnNumber</i> is returned in <i>NumericAttributePtr</i> . See Appendix D, “Data conversion”, on page 485 for a list of possible data type values that can be returned. For the datetime data types, this field returns the verbose data type, such as SQL_DATETIME.
SQL_DESC_TYPE_NAME (SQL_COLUMN_TYPE_NAME) ¹	<i>CharacterAttributePtr</i>	The type of the column (specified in an SQL statement) is returned in <i>CharacterAttributePtr</i> . See Appendix D, “Data conversion”, on page 485 for information on each data type.
SQL_DESC_UNNAMED ¹	<i>NumericAttributePtr</i>	SQL_NAMED or SQL_UNNAMED. If the SQL_DESC_NAME contains a column name, SQL_NAMED is returned. If there is no column name, SQL_UNNAMED is returned.
SQL_DESC_UNSIGNED (SQL_COLUMN_UNSIGNED) ¹	<i>NumericAttributePtr</i>	Indicates if the column data type is an unsigned type. SQL_TRUE is returned in <i>NumericAttributePtr</i> for all non-numeric data types. SQL_FALSE is returned for all numeric data types.

SQLColAttribute

Table 29. SQLColAttribute FieldIdentifiers (continued)

FieldIdentifier	Information returned in arguments	Description
SQL_DESC_UPDATABLE (SQL_COLUMN_UPDATABLE) ¹	NumericAttributePtr	Indicates if the column data type is an updateable data type: <ul style="list-style-type: none">• SQL_ATTR_READWRITE_UNKNOWN is returned in <i>NumericAttributePtr</i> for all DB2 SQL data types.• SQL_ATTR_READONLY is returned if the column is obtained from a catalog function call. ODBC also defines the following values, however DB2 ODBC does not return these values:<ul style="list-style-type: none">– SQL_DESC_UPDATABLE– SQL_UPDT_WRITE

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

Table 30. SQLColAttribute SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The buffer <i>*CharacterAttributePtr</i> was not large enough to return the entire string value, so the string value was truncated. The length of the untruncated string value is returned in <i>*StringLengthPtr</i> . (Function returns SQL_SUCCESS_WITH_INFO.)
07009	Invalid descriptor index.	The value specified for <i>ColumnNumber</i> was less than 0 or the value specified for the argument <i>ColumnNumber</i> was greater than the number of columns in the result set.
07005	The statement did not return a result set.	The statement associated with the <i>StatementHandle</i> did not return a result set. There were no columns to describe.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by SQLGetDiagRec() in the <i>*MessageText</i> buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC was not able to allocate memory support execution or completion of the function.
HY010	Function sequence error.	The function was called prior to calling SQLPrepare() or SQLExecDirect() for the <i>StatementHandle</i> . SQLExecute() or SQLExecDirect() was called for the <i>StatementHandle</i> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

1. The deprecated ODBC 2.0 SQLColAttribute() descriptor (argument fDescType) values. These values are supported in either the SQLColAttribute() or the SQLColAttribute() function.

Table 30. SQLColAttribute SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY090	Invalid string or buffer length.	The value specified for the argument <i>BufferLength</i> was less than 0.
HY091	Descriptor type out of range.	The value specified for the argument <i>FieldIdentifier</i> was not one of the defined values and was not an implementation-defined value.
HYC00	Driver not capable.	DB2 ODBC does not support the value specified for the argument <i>FieldIdentifier</i> .

Restrictions

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

Example

Sample to retrieve displaysize of first output column.

```
SQLINTEGER displaysize ;
SQLColAttribute( hstmt,
                 ( SQLSMALLINT ) ( 1 ),
                 SQL_DESC_DISPLAY_SIZE,
                 NULL,
                 0,
                 NULL,
                 &displaysize
                 ) ;
```

References

- “SQLBindCol - Bind a column to an application variable” on page 85
- “SQLDescribeCol - Describe column attributes” on page 137
- “SQLFetch - Fetch next row” on page 176
- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169

SQLColAttributes - Get column attributes

Purpose

Specification:	ODBC 1.0	X/OPEN CLI ¹	ISO CLI ¹
----------------	----------	-------------------------	----------------------

In ODBC 3.0, `SQLColAttribute()` replaces the ODBC 2.0 function `SQLColAttributes()`. See `SQLColAttribute()` for more information.

`SQLColAttributes()` is used to get an attribute for a column of the result set, and can also be used to determine the number of columns. `SQLColAttributes()` is a more extensible alternative to the `SQLDescribeCol()` function, but can only return one attribute per call.

Either `SQLPrepare()` or `SQLExecDirect()` must be called before calling this function.

If the application does not know the various attributes (such as, data type and length) of the column, this function (or `SQLDescribeCol()`) must be called before binding, using `SQLBindCol()`, to any columns.

Note: 1 - X/Open and ISO define this function with a singular name, `SQLColAttribute()`.

Syntax

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

Function arguments

Table 31. *SQLColAttributes* arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle
SQLUSMALLINT	<i>icol</i>	input	Column number in result set (must be between 1 and the number of columns in the results set inclusive). This argument is ignored when <code>SQL_COLUMN_COUNT</code> is specified.
SQLUSMALLINT	<i>fDescType</i>	input	The supported values are described in Table 32 on page 115.
SQLCHAR *	<i>rgbDesc</i>	output	Pointer to buffer for string column attributes
SQLSMALLINT	<i>cbDescMax</i>	input	Length of <i>rgbDesc</i> descriptor buffer.
SQLSMALLINT *	<i>pcbDesc</i>	output	Actual number of bytes returned in <i>rgbDesc</i> buffer. If this argument contains a value equal to or greater than the length specified in <i>cbDescMax</i> , truncation occurred. The column attribute value is then truncated to <i>cbDescMax</i> bytes minus the size of the null-terminator (or to <i>cbDescMax</i> bytes if null termination is off).
SQLINTEGER *	<i>pfDesc</i>	output	Pointer to integer which holds the value of numeric column attributes.

The following values can be specified for the *fDescType* argument:

Table 32. *fdescType* descriptor types

Descriptor	Description
SQL_COLUMN_AUTO_INCREMENT	<p>Indicates if the column data type is an auto increment data type.</p> <p>FALSE is returned in <i>pfDesc</i> for all DB2 SQL data types.</p>
SQL_COLUMN_CASE_SENSITIVE	<p>Indicates if the column data type is a case sensitive data type.</p> <p>Either TRUE or FALSE is returned in <i>pfDesc</i> depending on the data type.</p> <p>Case sensitivity does not apply to graphic data types, FALSE is returned.</p> <p>FALSE is returned for non-character data types.</p>
SQL_COLUMN_CATALOG_NAME (SQL_COLUMN_QUALIFIER_NAME)	<p>The catalog of the table that contains the column is returned in <i>rgbDesc</i>. An empty string is returned since DB2 ODBC only supports two-part naming for a table.</p> <p>SQL_COLUMN_QUALIFIER_NAME is defined for compatibility with ODBC. DB2 ODBC applications should use SQL_COLUMN_CATALOG_NAME.</p>
SQL_COLUMN_COUNT	<p>The number of columns in the result set is returned in <i>pfDesc</i>.</p>
SQL_COLUMN_DISPLAY_SIZE	<p>The maximum number of bytes needed to display the data in character form is returned in <i>pfDesc</i>.</p> <p>See Table 177 on page 489 for the display size of each of the column types.</p>
SQL_COLUMN_DISTINCT_TYPE	<p>The distinct type data type name of the column is returned in <i>rgbDesc</i>. If the column is a built-in SQL type and not a distinct type, an empty string is returned.</p> <p>Note: This is an IBM-defined extension to the list of descriptor attributes defined by ODBC.</p>
SQL_COLUMN_LABEL	<p>The column label is returned in <i>rgbDesc</i>. If the column does not have a label, the column name or the column expression is returned. If the column is unlabeled and unnamed, an empty string is returned.</p>

SQLColAttributes

Table 32. *fdesc*type descriptor types (continued)

Descriptor	Description
SQL_COLUMN_LENGTH	<p>The number of <i>bytes</i> of data associated with the column is returned in <i>pfDesc</i>. This is the length in bytes of data transferred on the fetch or <code>SQLGetData()</code> for this column if <code>SQL_C_DEFAULT</code> is specified as the C data type. See Table 176 on page 488 for the length of each of the SQL data types.</p> <p>If the column identified in <i>icol</i> is a fixed length character or binary string, (for example, <code>SQL_CHAR</code> or <code>SQL_BINARY</code>) the actual length is returned.</p> <p>If the column identified in <i>icol</i> is a variable length character or binary string, (for example, <code>SQL_VARCHAR</code>) the maximum length is returned.</p>
SQL_COLUMN_MONEY	<p>Indicates if the column data type is a money data type.</p> <p>FALSE is returned in <i>pfDesc</i> for all DB2 SQL data types.</p>
SQL_COLUMN_NAME	<p>The name of the column <i>icol</i> is returned in <i>rgbDesc</i>. If the column is an expression, then the result returned is product specific.</p>
SQL_COLUMN_NULLABLE	<p>If the column identified by <i>icol</i> can contain nulls, then <code>SQL_NULLABLE</code> is returned in <i>pfDesc</i>.</p> <p>If the column is constrained not to accept nulls, then <code>SQL_NO_NULLS</code> is returned in <i>pfDesc</i>.</p>
SQL_COLUMN_PRECISION	<p>The precision in units of digits is returned in <i>pfDesc</i> if the column is <code>SQL_DECIMAL</code>, <code>SQL_NUMERIC</code>, <code>SQL_DOUBLE</code>, <code>SQL_FLOAT</code>, <code>SQL_INTEGER</code>, <code>SQL_REAL</code> or <code>SQL_SMALLINT</code>.</p> <p>If the column is a character SQL data type, then the precision returned in <i>pfDesc</i>, indicates the maximum number of <i>characters</i> the column can hold.</p> <p>If the column is a graphic SQL data type, then the precision returned in <i>pfDesc</i>, indicates the maximum number of double-byte <i>characters</i> the column can hold.</p> <p>See Table 174 on page 486 for the precision of each of the SQL data types.</p>
SQL_COLUMN_SCALE	<p>The scale attribute of the column is returned. See Table 175 on page 487 for the scale of each of the SQL data types.</p>
SQL_COLUMN_SCHEMA_NAME (SQL_COLUMN_OWNER_NAME)	<p>The schema of the table that contains the column is returned in <i>rgbDesc</i>. An empty string is returned as DB2 ODBC is unable to determine this attribute.</p> <p><code>SQL_COLUMN_OWNER_NAME</code> is defined for compatibility with ODBC. DB2 ODBC applications should use <code>SQL_COLUMN_SCHEMA_NAME</code>.</p>

Table 32. *fdesc*type descriptor types (continued)

Descriptor	Description
SQL_COLUMN_SEARCHABLE	Indicates if the column data type is searchable: <ul style="list-style-type: none"> • SQL_UNSEARCHABLE if the column cannot be used in a WHERE clause. • SQL_LIKE_ONLY if the column can be used in a WHERE clause only with the LIKE predicate. • SQL_ALL_EXCEPT_LIKE if the column can be used in a WHERE clause with all comparison operators except LIKE. • SQL_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.
SQL_COLUMN_TABLE_NAME	The name of the table that contains the column is returned in <i>rgbDesc</i> . An empty string is returned as DB2 ODBC cannot determine this attribute.
SQL_COLUMN_TYPE	The SQL data type of the column identified in <i>icol</i> is returned in <i>pfDesc</i> . The possible values for <i>pfSqlType</i> are listed in Table 4 on page 31.
SQL_COLUMN_TYPE_NAME	The type of the column (as entered in an SQL statement) is returned in <i>rgbDesc</i> . For information on each data type see the TYPE_NAME attribute found in “Data types and data conversion” on page 30.
SQL_COLUMN_UNSIGNED	Indicates if the column data type is an unsigned type or not. TRUE is returned in <i>pfDesc</i> for all non-numeric data types, FALSE is returned for all numeric data types.
SQL_COLUMN_UPDATABLE	Indicates if the column data type is an updateable data type. SQL_ATTR_READWRITE_UNKNOWN is returned in <i>pfDesc</i> for all DB2 SQL data types. SQL_ATTR_READONLY is returned if the column is obtained from a catalog function call.

Usage

Instead of returning a specific set of attributes like `SQLDescribeCol()`, `SQLColAttributes()` allows you to specify which attribute you wish to receive for a specific column. If the desired information is a string, it is returned in *rgbDesc*. If the desired information is a number, it is returned in *pfDesc*.

`SQLColAttributes()` is an extensible alternative to `SQLDescribeCol()`, which is used to return a fixed set of commonly used column attribute information.

If an *fDescType* descriptor type does not apply to the database server, an empty string is returned in *rgbDesc* or zero is returned in *pfDesc*, depending on the expected result of the descriptor.

Columns are identified by a number (numbered sequentially from left to right starting with 1) and can be described in any order.

SQLColAttributes

Calling `SQLColAttributes()` with `fDescType` set to `SQL_COLUMN_COUNT` is an alternative to calling `SQLNumResultCols()` to determine whether any columns can be returned.

Return codes

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`

Diagnostics

Table 33. `SQLColAttributes` `SQLSTATEs`

SQLSTATE	Description	Explanation
01004	Data truncated.	The character string returned in the argument <i>rgbDesc</i> is longer than the value specified in the argument <i>cbDescMax</i> . The argument <i>pcbDesc</i> contains the actual length of the string to be returned. (Function returns <code>SQL_SUCCESS_WITH_INFO</code> .)
07005	The statement did not return a result set.	The statement associated with the <i>hstmt</i> did not return a result set. There are no columns to describe. To prevent encountering this error, call <code>SQLNumResultCols()</code> before calling <code>SQLColAttributes()</code> .
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1002	Invalid column number.	The value specified for the argument <i>icol</i> is less than 1. The value specified for the argument <i>icol</i> is greater than the number of columns in the result set. Not returned if <code>SQL_COLUMN_COUNT</code> is specified.
S1010	Function sequence error.	The function is called prior to calling <code>SQLPrepare()</code> or <code>SQLExecuteDirect()</code> for the <i>hstmt</i> . The function is called while in a data-at-execute (<code>SQLParamData()</code> , <code>SQLPutData()</code>) operation.
S1013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
S1090	Invalid string or buffer length.	The length specified in the argument <i>cbDescMax</i> is less than 0 and <i>fDescType</i> requires a character string be returned in <i>rgbDesc</i> .
S1091	Descriptor type out of range.	The value specified for the argument <i>fDescType</i> was not equal to a value specified in Table 32 on page 115.
S1C00	Driver not capable.	The SQL data type returned by the database server for column <i>icol</i> is not recognized by DB2 ODBC.

Restrictions

None.

Example

See “Example” on page 139

References

- “SQLDescribeCol - Describe column attributes” on page 137
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLPrepare - Prepare a statement” on page 300
- “SQLSetColAttributes - Set column attributes” on page 332

SQLColumnPrivileges - Get privileges associated with the columns of a table

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLColumnPrivileges() returns a list of columns and associated privileges for the specified table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated from a query.

Syntax

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

Function arguments

Table 34. SQLColumnPrivileges arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szCatalogName	input	Catalog qualifier of a 3-part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbCatalogName	input	Length of <i>szCatalogName</i> . This must be set to 0.
SQLCHAR *	szSchemaName	input	Schema qualifier of table name.
SQLSMALLINT	cbSchemaName	input	Length of <i>szSchemaName</i> .
SQLCHAR *	szTableName	input	Table name.
SQLSMALLINT	cbTableName	input	Length of <i>szTableName</i> .
SQLCHAR *	szColumnName	input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by column name.
SQLSMALLINT	cbColumnName	input	Length of <i>szColumnName</i> .

Usage

The results are returned as a standard result set containing the columns listed in Table 35 on page 121. The result set is ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME, and PRIVILEGE. If multiple privileges are associated with any given column, each privilege is returned as a separate row. A typical application might wish to call this function after a call to SQLColumns() to determine column privilege information. The application should use the character strings returned in the TABLE_SCHEM, TABLE_NAME, COLUMN_NAME columns of the SQLColumns() result set as input arguments to this function.

Since calls to `SQLColumnPrivileges()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating the calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the `SQL_MAX_CATALOG_NAME_LEN`, `SQL_MAX_SCHEMA_NAME_LEN`, `SQL_MAX_TABLE_NAME_LEN`, and `SQL_MAX_COLUMN_NAME_LEN` to determine respectively the actual lengths of the `TABLE_CAT`, `TABLE_SCHEM`, `TABLE_NAME`, and `COLUMN_NAME` columns supported by the connected DBMS.

Note that the `szColumnName` argument accepts a search pattern. For more information about valid search patterns, see “Input arguments on catalog functions” on page 398.

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 35. Columns returned by `SQLColumnPrivileges`

Column number/name	Data type	Description
1 TABLE_CAT	VARCHAR(128)	This is always NULL.
2 TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
3 TABLE_NAME	VARCHAR(128) not NULL	Name of the table or view.
4 COLUMN_NAME	VARCHAR(128) not NULL	Name of the column of the specified table or view.
5 GRANTOR	VARCHAR(128)	Authorization ID of the user who granted the privilege.
6 GRANTEE	VARCHAR(128)	Authorization ID of the user to whom the privilege is granted.
7 PRIVILEGE	VARCHAR(128)	The column privilege. This can be: <ul style="list-style-type: none"> • ALTER • CONTROL • DELETE • INDEX • INSERT • REFERENCES • SELECT • UPDATE

Supported privileges are based on the data source to which you are connected.

Note: Most IBM RDBMSs do not offer column level privileges at the column level. DB2 for OS/390 and z/OS and DB2 for VSE & VM support the UPDATE column privilege; there is one row in this result set for each updateable column. For all other privileges for DB2 for OS/390 and z/OS and DB2 for VSE & VM, and for all privileges for other IBM RDBMSs, if a privilege has been granted at the table level, a row is present in this result set.

SQLColumnPrivileges

Table 35. Columns returned by SQLColumnPrivileges (continued)

Column number/name	Data type	Description
8 IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users. Either "YES", "NO".

Note: 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 SQLColumnPrivileges() result set in ODBC.

If there is more than one privilege associated with a column, then each privilege is returned as a separate row in the result set.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 36. SQLColumnPrivileges SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	<i>szTableName</i> is a null.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal to SQL_NTS.
HYC00	Driver not capable.	DB2 ODBC does not support <i>catalog</i> as a qualifier for table name.

Restrictions

None.

Example

```

/* ... */
SQLRETURN
list_column_privileges(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */

    rc = SQLColumnPrivileges(hstmt, NULL, 0, schema, SQL_NTS,
                             tablename, SQL_NTS, columnname.s, SQL_NTS);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) columnname.s, 129,
                    &columnname.ind);

    rc = SQLBindCol(hstmt, 5, SQL_C_CHAR, (SQLPOINTER) grantor.s, 129,
                    &grantor.ind);

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR, (SQLPOINTER) grantee.s, 129,
                    &grantee.ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) privilege.s, 129,
                    &privilege.ind);

    rc = SQLBindCol(hstmt, 8, SQL_C_CHAR, (SQLPOINTER) is_grantable.s, 4,
                    &is_grantable.ind);

    printf("Column Privileges for %s.%s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        sprintf(cur_name, "Column: %s\n", columnname.s);
        if (strcmp(cur_name, pre_name) != 0) {
            printf("\n%s\n", cur_name);
            printf("  Grantor      Grantee      Privilege  Grantable\n");
            printf("  -----  -----  -----  ---\n");
        }
        strcpy(pre_name, cur_name);
        printf("    %-15s", grantor.s);
        printf("    %-15s", grantee.s);
        printf("    %-10s", privilege.s);
        printf("    %-3s\n", is_grantable.s);
    }
    /* endwhile */
/* ... */

```

References

- “SQLColumns - Get column information for a table” on page 124
- “SQLTables - Get table information” on page 382

SQLColumns - Get column information for a table

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	
-----------------------	-----------------	-------------------	--

SQLColumns() returns a list of columns in the specified tables. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to fetch a result set generated by a query.

Syntax

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

Function arguments

Table 37. SQLColumns arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szCatalogName	input	Buffer that can contain a <i>pattern-value</i> to qualify the result set. <i>Catalog</i> is the first part of a 3-part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbCatalogName	input	Length of <i>szCatalogName</i> . This must be set to 0.
SQLCHAR *	szSchemaName	input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by schema name.
SQLSMALLINT	cbSchemaName	input	Length of <i>szSchemaName</i> .
SQLCHAR *	szTableName	input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by table name.
SQLSMALLINT	cbTableName	input	Length of <i>szTableName</i> .
SQLCHAR *	szColumnName	input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by column name.
SQLSMALLINT	cbColumnName	input	Length of <i>szColumnName</i> .

Usage

This function is called to retrieve information about the columns of either a table or a set of tables. A typical application might wish to call this function after a call to SQLTables() to determine the columns of a table. The application should use the character strings returned in the TABLE_SCHEMA and TABLE_NAME columns of the SQLTables() result set as input to this function.

SQLColumns() returns a standard result set, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and ORDINAL_POSITION. Table 38 on page 125 lists the columns in the result set.

The *szSchemaName*, *szTableName*, and *szColumnName* arguments accept search patterns. For more information about valid search patterns, see “Input arguments on catalog functions” on page 398.

Since calls to `SQLColumns()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the `SQL_MAX_CATALOG_NAME_LEN`, `SQL_MAX_OWNER_SCHEMA_LEN`, `SQL_MAX_TABLE_NAME_LEN`, and `SQL_MAX_COLUMN_NAME_LEN` to determine respectively the actual lengths of the `TABLE_CAT`, `TABLE_SCHEM`, `TABLE_NAME`, and `COLUMN_NAME` columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 38. Columns returned by `SQLColumns`

Column number/name	Data type	Description
1 TABLE_CAT	VARCHAR(128)	This is always NULL.
2 TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
3 TABLE_NAME	VARCHAR(128) NOT NULL	Name of the table, view, alias, or synonym.
4 COLUMN_NAME	VARCHAR(128) NOT NULL	Column identifier. Name of the column of the specified table, view, alias, or synonym.
5 DATA_TYPE	SMALLINT NOT NULL	SQL data type of column identified by COLUMN_NAME. This is one of the values in the Symbolic SQL Data Type column in Table 4 on page 31.
6 TYPE_NAME	VARCHAR(128) NOT NULL	Character string representing the name of the data type corresponding to DATA_TYPE.
7 COLUMN_SIZE	INTEGER	If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in characters for the column. For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set. See also, Table 174 on page 486.
8 BUFFER_LENGTH	INTEGER	The maximum number of bytes for the associated C buffer to store data from this column if <code>SQL_C_DEFAULT</code> is specified on the <code>SQLBindCol()</code> , <code>SQLGetData()</code> and <code>SQLBindParameter()</code> calls. This length does not include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign. See also, Table 176 on page 488.

SQLColumns

Table 38. Columns returned by SQLColumns (continued)

Column number/name	Data type	Description
9 DECIMAL_DIGITS	SMALLINT	<p>The scale of the column. NULL is returned for data types where scale is not applicable.</p> <p>See also, Table 175 on page 487.</p>
10 NUM_PREC_RADIX	SMALLINT	<p>Either 10 or 2 or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, then the COLUMN_SIZE column contains the number of bits allowed in the column.</p> <p>If DATA_TYPE is an exact numeric data type, this column contains the value 10 and the COLUMN_SIZE contains the number of decimal digits allowed for the column.</p> <p>For numeric data types, the DBMS can return a NUM_PREC_RADIX of either 10 or 2.</p> <p>NULL is returned for data types where radix is not applicable.</p>
11 NULLABLE	SMALLINT NOT NULL	<p>SQL_NO_NULLS if the column does not accept NULL values.</p> <p>SQL_NULLABLE if the column accepts NULL values.</p>
12 REMARKS	VARCHAR(254)	Might contain descriptive information about the column.
13 COLUMN_DEF	VARCHAR(254)	<p>The column's default value. If the default value is a numeric literal, then this column contains the character representation of the numeric literal with no enclosing single quotes. If the default value is a character string, then this column is that string enclosed in single quotes. If the default value is a <i>pseudo-literal</i>, such as for DATE, TIME, and TIMESTAMP columns, then this column contains the keyword of the pseudo-literal (for example, CURRENT DATE) with no enclosing quotes.</p> <p>If NULL was specified as the default value, then this column returns the word NULL, not enclosed in quotes. If the default value cannot be represented without truncation, then this column contains TRUNCATED with no enclosing single quotes. If no default value was specified, then this column is NULL.</p>
14 SQL_DATA_TYPE	SMALLINT NOT NULL	<p>The SQL data type. This column is the same as the DATA_TYPE column. For datetime data types, the SQL_DATA_TYPE field in the result set is SQL_DATETIME, and the SQL_DATETIME_SUB field returns the subcode for the specific datetime data type (SQL_CODE_DATE, SQL_CODE_TIME or SQL_CODE_TIMESTAMP).</p>
15 SQL_DATETIME_SUB	SMALLINT	<p>The subtype code for datetime data types:</p> <ul style="list-style-type: none"> • SQL_CODE_DATE • SQL_CODE_TIME • SQL_CODE_TIMESTAMP <p>For all other data types, this column returns a NULL.</p>

Table 38. Columns returned by SQLColumns (continued)

Column number/name	Data type	Description
16 CHAR_OCTET_LENGTH	INTEGER	Contains the maximum length in octets for a character data type column. For Single Byte character sets, this is the same as COLUMN_SIZE. For all other data types it is NULL.
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.

Note: This result set is identical to the X/Open CLI Columns() result set specification, which is an extended version of the SQLColumns() result set specified in ODBC V2. The ODBC SQLColumns() result set includes every column in the same position up to the REMARKS column.

DB2 ODBC applications that issue SQLColumns() against a DB2 for OS/390 and z/OS server, Version 5 or later, should expect the result set columns listed in the table above. Revision bars identify the new and changed columns.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 39. SQLColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal SQL_NTS.
HYC00	Driver not capable.	DB2 ODBC does not support <i>catalog</i> as a qualifier for table name.

Restrictions

None.

SQLColumns

Example

```
/* ... */
SQLRETURN
list_columns(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */

    rc = SQLColumns(hstmt, NULL, 0, schema, SQL_NTS,
                   tablename, SQL_NTS, "%", SQL_NTS);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) column_name.s, 129,
                   &column_name.ind);

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR, (SQLPOINTER) type_name.s, 129,
                   &type_name.ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_LONG, (SQLPOINTER) &length,
                   sizeof(length), &length_ind);

    rc = SQLBindCol(hstmt, 9, SQL_C_SHORT, (SQLPOINTER) &scale,
                   sizeof(scale), &scale_ind);

    rc = SQLBindCol(hstmt, 12, SQL_C_CHAR, (SQLPOINTER) remarks.s, 129,
                   &remarks.ind);

    rc = SQLBindCol(hstmt, 11, SQL_C_SHORT, (SQLPOINTER) &nullable,
                   sizeof(nullable), &nullable_ind);

    printf("Schema: %s Table Name: %s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf(" %s", column_name.s);
        if (nullable == SQL_NULLABLE) {
            printf(", NULLABLE");
        } else {
            printf(", NOT NULLABLE");
        }
        printf(", %s", type_name.s);
        if (length_ind != SQL_NULL_DATA) {
            printf(" (%ld", length);
        } else {
            printf("\n");
        }
        if (scale_ind != SQL_NULL_DATA) {
            printf(", %d)\n", scale);
        } else {
            printf("\n");
        }
    }
    /* endwhile */
/* ... */
```

References

- “SQLTables - Get table information” on page 382
- “SQLColumnPrivileges - Get privileges associated with the columns of a table” on page 120
- “SQLSpecialColumns - Get special (row identifier) columns” on page 369

SQLConnect - Connect to a data source

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLConnect() establishes a connection to the target database. The application must supply a target SQL database.

A connection handle must be allocated using SQLAllocHandle() before calling this function.

This function must be called before allocating a statement handle using SQLAllocHandle().

Syntax

```
SQLRETURN SQLConnect(
    (SQLHDBC          hdbc,
     SQLCHAR          FAR *szDSN,
     SQLSMALLINT      cbDSN,
     SQLCHAR          FAR *szUID,
     SQLSMALLINT      cbUID,
     SQLCHAR          FAR *szAuthStr,
     SQLSMALLINT      cbAuthStr);
```

Function arguments

Table 40. SQLConnect arguments

Data type	Argument	Use	Description
SQLHDBC	<i>hdbc</i>	input	Connection handle
SQLCHAR *	<i>szDSN</i>	input	Data Source: The name or alias-name of the database.
SQLSMALLINT	<i>cbDSN</i>	input	Length of contents of <i>szDSN</i> argument.
SQLCHAR *	<i>szUID</i>	input	Authorization-name (user identifier). This parameter is validated and authenticated.
SQLSMALLINT	<i>cbUID</i>	input	Length of contents of <i>szUID</i> argument. This parameter is validated and authenticated.
SQLCHAR *	<i>szAuthStr</i>	input	Authentication-string (password). This parameter is validated and authenticated.
SQLSMALLINT	<i>cbAuthStr</i>	input	Length of contents of <i>szAuthStr</i> argument. This parameter is validated and authenticated.

Usage

The target database (also known as *data source*) for IBM RDBMSs is the location name as defined in SYSIBM.LOCATIONS when DDF has been configured in the DB2 subsystem. The application can obtain a list of databases available to connect to by calling SQLDataSources(). For many applications, a local database is being accessed (DDF is not being used). 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 the application can use a 'null connect', as described below, to connect to the default local database without supplying a database name.

SQLConnect

The input length arguments to `SQLConnect()` (*cbDSN*, *cbUID*, *cbAuthStr*) can be set to the actual length of their associated data (not including any null-terminating character) or to `SQL_NTS` to indicate that the associated data is null-terminated.

```
# If a user ID is specified on SQLConnect(), the user ID (szuid) and a password  
# (szauthstr) are passed to the target data source for authentication. szuid and  
# szauthstr must not contain any blanks. If the user ID is null or empty, authentication  
# is not performed.
```

```
# The user is authenticated when argument values are specified on SQLConnect() for  
# both a user ID (szuid) and a password (szauthstr). When authentication is  
# performed, a CONNECT statement is run. The CONNECT statement cannot run if  
# the user ID is null or empty. szuid and szauthstr must not contain any blanks.
```

The semantics of *szDSN* are as follows:

- If the *szDSN* argument pointer is `NULL` or the *cbDSN* argument value is 0, this is a null `SQLConnect()`. (The function performed by a null `SQLConnect()` is referred to as a 'null connect'.)

A null `SQLConnect()` still requires that `SQLAllocHandle()` with *HandleType* `SQL_HANDLE_ENV` and `SQLAllocHandle()` with *HandleType* `SQL_HANDLE_DBC` be called first. Reasons for coding a null `SQLConnect()` include:

- The DB2 ODBC application needs to connect to the default data source. (The default data source is the DB2 subsystem specified by the `MVSDEFAULTSSID` initialization file setting.)
 - The DB2 ODBC application is mixing embedded SQL and DB2 ODBC calls, and the application already connected to a data source before invoking DB2 ODBC. In this case, the application must issue a null `SQLConnect()`.
 - The DB2 ODBC application is running as a stored procedure. DB2 ODBC applications running as stored procedures must issue a null `SQLConnect()`.
- If the *szDSN* argument pointer is not `NULL` and the *cbDSN* argument value is not 0, DB2 ODBC issues a `CONNECT` to the data source.

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.

Various connection characteristics (options) can be specified by the end user in the section of the initialization file associated with the *szDSN* data source argument or set by the application using `SQLSetConnectAttr()`. The extended connect function, `SQLDriverConnect()`, can be called with additional connect options and can also perform a null connect.

For a null `SQLConnect()`, the `CONNECT` type defaults to the value of the `CONNECTTYPE` keyword specified in the common section of the initialization file. The DB2 ODBC application can override the `CONNECT` type by specifying the parameter of the `SQL_CONNECTTYPE` option using:

- `SQLSetConnectAttr()` before the null `SQLConnect()` is issued, or
- `SQLSetEnvAttr` before the null `SQLConnect()` is issued.

A connection established by `SQLConnect()` recognizes externally created contexts and allows multiple connections to the same data source from different contexts.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 41. SQLConnect SQLSTATES

SQLSTATE	Description	Explanation
08001	Unable to connect to data source.	DB2 ODBC is not able to establish a connection with the data source (server). The connection request is rejected because an existing connection established using embedded SQL already exists.
08002	Connection in use.	The specified <i>hdbc</i> is already being used to establish a connection with a data source and the connection is still open.
08004	The application server rejected establishment of the connection.	The data source (server) rejected the establishment of the connection. The number of connections specified by the MAXCONN keyword has been reached.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY024	Invalid argument value.	A non-matching double quote (") is found in either the <i>szDSN</i> , <i>szUID</i> , or <i>szAuthStr</i> argument.
HY090	Invalid string or buffer length.	The value specified for argument <i>cbDSN</i> is less than 0, but not equal to SQL_NTS and the argument <i>szDSN</i> is not a null pointer. The value specified for argument <i>cbUID</i> is less than 0, but not equal to SQL_NTS and the argument <i>szUID</i> is not a null pointer. The value specified for argument <i>cbAuthStr</i> is less than 0, but not equal to SQL_NTS and the argument <i>szAuthStr</i> is not a null pointer.
#	#	The authorization name specified for argument <i>szUID</i> exceeded the maximum length supported.
#	#	The authentication string specified for argument <i>szAuthStr</i> exceeded the maximum length supported.
S1501	Invalid data source name.	An invalid data source name is specified in argument <i>szDSN</i> .

Restrictions

The implicit connection (or default database) option for IBM RDBMSs is not supported. SQLConnect () must be called before any SQL statements can be executed.

Example

```

/* ... */
/* 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);
    }
    /* Pass user ID and password to target server for authentication */
    rc = SQLConnect(*hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf(">--- Error while connecting to database: %s -----\n", server);
        SQLDisconnect(*hdbc);
        SQLFreeHandle (SQL_HANDLE_DBC, *hdbc);
        return (SQL_ERROR);
    } else {
        /* Print Connection Information */
        printf(">Connected to %s\n", server);
    }
    return (SQL_SUCCESS);
}

/*****
/* DBconnect2 - Connect with connect type */
/* Valid connect types SQL_CONCURRENT_TRANS, SQL_COORDINATED_TRANS */
/*****
SQLRETURN DBconnect2(SQLHENV henv,
                     SQLHDBC * hdbc, SQLINTEGER contype)
                     SQLHDBC * hdbc, 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: %s -----\n", server);
    SQLDisconnect(*hdbc);
    SQLFreeHandle(SQL_HANDLE_DBC, *hdbc)
    return (SQL_ERROR);
} else {
    /* Print Connection Information */
    printf(">Connected to %s\n", server);
}
return (SQL_SUCCESS);
}
/* ... */

```

References

- “SQLAllocHandle - Allocate handle” on page 79
- “SQLSetConnectOption - Set connection option” on page 345
- “SQLDataSources - Get list of data sources” on page 134
- “SQLDisconnect - Disconnect from a data source” on page 144
- “SQLDriverConnect - (Expanded) connect to a data source” on page 146
- “SQLGetConnectOption - Returns current setting of a connect option” on page 202

SQLDataSources - Get list of data sources

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

SQLDataSources() returns a list of target databases available, one at a time.

SQLDataSources() is usually called before a connection is made, to determine the databases that are available to connect to.

Syntax

```
SQLRETURN SQLDataSources (SQLHENV          henv,
                          SQLUSMALLINT     fDirection,
                          SQLCHAR          FAR *szDSN,
                          SQLSMALLINT      cbDSNMax,
                          SQLSMALLINT FAR *pcbDSN,
                          SQLCHAR          FAR *szDescription,
                          SQLSMALLINT      cbDescriptionMax,
                          SQLSMALLINT FAR *pcbDescription);
```

Function arguments

Table 42. SQLDataSources arguments

Data Type	Argument	Use	Description
SQLHENV	<i>henv</i>	input	Environment handle.
SQLUSMALLINT	<i>fDirection</i>	input	Used by application to request the first data source name in the list or the next one in the list. <i>fDirection</i> can take on only the following values: <ul style="list-style-type: none"> SQL_FETCH_FIRST SQL_FETCH_NEXT
SQLCHAR *	<i>szDSN</i>	output	Pointer to buffer to hold the data source name retrieved.
SQLSMALLINT	<i>cbDSNMax</i>	input	Maximum length of the buffer pointed to by <i>szDSN</i> . This should be less than or equal to SQL_MAX_DSN_LENGTH + 1.
SQLSMALLINT *	<i>pcbDSN</i>	output	Pointer to location where the maximum number of bytes available to return in the <i>szDSN</i> are stored.
SQLCHAR *	<i>szDescription</i>	output	Pointer to buffer where the description of the data source is returned. DB2 ODBC returns the Comment field associated with the database cataloged to the DBMS.
Note: IBM RDBMSs always return blank padded to 30 bytes.			
SQLSMALLINT	<i>cbDescriptionMax</i>	input	Maximum length of the <i>szDescription</i> buffer. DB2 for OS/390 and z/OS always returns NULL.
SQLSMALLINT *	<i>pcbDescription</i>	output	Pointer to location where this function returns the actual number of bytes available to return for the description of the data source. DB2 for OS/390 and z/OS always returns zero.

Usage

The application can call this function any time with *fDirection* set to either SQL_FETCH_FIRST or SQL_FETCH_NEXT.

If `SQL_FETCH_FIRST` is specified, the first database in the list is always returned.

If `SQL_FETCH_NEXT` is specified:

- Directly following a `SQL_FETCH_FIRST` call, the second database in the list is returned
- Before any other `SQLDataSources()` call, the first database in the list is returned
- When there are no more databases in the list, `SQL_NO_DATA_FOUND` is returned. If the function is called again, the first database is returned.
- Any other time, the next database in the list is returned.

Return codes

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`
- `SQL_NO_DATA_FOUND`

Diagnostics

Table 43. *SQLDataSources* SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data source name returned in the argument <i>szDSN</i> is longer than the value specified in the argument <i>cbDSNMax</i> . The argument <i>pcbDSN</i> contains the length of the full data source name. (Function returns <code>SQL_SUCCESS_WITH_INFO</code> .)
		The data source name returned in the argument <i>szDescription</i> is longer than the value specified in the argument <i>cbDescriptionMax</i> . The argument <i>pcbDescription</i> contains the length of the full data source description. (Function returns <code>SQL_SUCCESS_WITH_INFO</code> .)
58004	Unexpected system failure.	Unrecoverable system error.
HY000	General error.	An error occurred for which there is no specific SQLSTATE and for which no specific SQLSTATE is defined. The error message returned by <code>SQLGetDiagRec()</code> in the argument <i>szErrorMsg</i> describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument <i>cbDSNMax</i> is less than 0.
		The value specified for argument <i>cbDescriptionMax</i> is less than 0.
HY103	Direction option out of range.	The value specified for the argument <i>fDirection</i> is not equal to <code>SQL_FETCH_FIRST</code> or <code>SQL_FETCH_NEXT</code> .

Restrictions

None.

SQLDataSources

Example

```
/* ... */
/*****
**   - 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    buff1, des1;

    /* 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, &buff1, description, 255, &des1))
           != SQL_NO_DATA_FOUND) {
        printf("%-30s %s\n", source, description);
    }

    SQLFreeHandle (SQL_HANDLE_ENV, henv);

    return (SQL_SUCCESS);
}
/* ... */
```

References

None.

SQLDescribeCol - Describe column attributes

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLDescribeCol() returns a set of commonly used descriptor information (column name, type, precision, scale, nullability) for the indicated column in the result set generated by a query.

If the application needs only one attribute of the descriptor information, or needs an attribute not returned by SQLDescribeCol(), the SQLColAttribute() function can be used in place of SQLDescribeCol(). See “SQLColAttributes - Get column attributes” on page 114 for more information.

Either SQLPrepare() or SQLExecDirect() must be called before calling this function.

This function (or SQLColAttribute()) is usually called before a bind column function SQLBindCol() to determine the attributes of a column before binding it to an application variable.

Syntax

```
SQLRETURN SQLDescribeCol (SQLHSTMT hstmt,
                          SQLUSMALLINT icol,
                          SQLCHAR FAR *szColName,
                          SQLSMALLINT cbColNameMax,
                          SQLSMALLINT FAR *pcbColName,
                          SQLSMALLINT FAR *pfSqlType,
                          SQLUINTEGER FAR *pcbColDef,
                          SQLSMALLINT FAR *pibScale,
                          SQLSMALLINT FAR *pfNullable);
```

Function arguments

Table 44. SQLDescribeCol arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle.
SQLUSMALLINT	<i>icol</i>	input	Column number to be described. Columns are numbered sequentially from left to right, starting at 1.
SQLCHAR *	<i>szColName</i>	output	Pointer to column name buffer. Set this to NULL if column name is not needed.
SQLSMALLINT	<i>cbColNameMax</i>	input	Size of <i>szColName</i> buffer.
SQLSMALLINT *	<i>pcbColName</i>	output	Bytes available to return for <i>szColName</i> argument. Truncation of column name (<i>szColName</i>) to <i>cbColNameMax</i> - 1 bytes occurs if <i>pcbColName</i> is greater than or equal to <i>cbColNameMax</i> .
SQLSMALLINT *	<i>pfSqlType</i>	output	Base SQL data type of column. To determine if there is a distinct type associated with the column, call SQLColAttribute() with <i>fDescType</i> set to SQL_COLUMN_DISTINCT_TYPE. See the symbolic SQL data type column of Table 4 on page 31 for the data types that are supported.

SQLDescribeCol

Table 44. SQLDescribeCol arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER *	<i>pcbColDef</i>	output	Precision of column as defined in the database. If <i>fSqlType</i> denotes a graphic or DBCLOB SQL data type, then this variable indicates the maximum number of double-byte characters the column can hold.
SQLSMALLINT *	<i>pibScale</i>	output	Scale of column as defined in the database (only applies to SQL_DECIMAL, SQL_NUMERIC, SQL_TYPE_TIMESTAMP). See Table 175 on page 487 for the scale of each of the SQL data types.
SQLSMALLINT *	<i>pfNullable</i>	output	Indicates whether NULLS are allowed for this column <ul style="list-style-type: none">• SQL_NO_NULLS• SQL_NULLABLE

Usage

Columns are identified by a number, are numbered sequentially from left to right starting with 1, and can be described in any order.

If a null pointer is specified for any of the pointer arguments, DB2 ODBC assumes that the information is not needed by the application and nothing is returned.

If the column is a distinct type, SQLDescribeCol() only returns the built-in type in *pfSqlType*. Call SQLColAttribute() with *fDescType* set to SQL_COLUMN_DISTINCT_TYPE to obtain the distinct type.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

If SQLDescribeCol() returns either SQL_ERROR, or SQL_SUCCESS_WITH_INFO, one of the following SQLSTATEs can be obtained by calling the SQLGetDiagRec() function.

Table 45. SQLDescribeCol SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The column name returned in the argument <i>szColName</i> is longer than the value specified in the argument <i>cbColNameMax</i> . The argument <i>pcbColName</i> contains the length of the full column name. (Function returns SQL_SUCCESS_WITH_INFO.)
07005	The statement did not return a result set.	The statement associated with the <i>hstmt</i> did not return a result set. There are no columns to describe. (Call SQLNumResultCols() first to determine if there are any rows in the result set.)
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.

Table 45. SQLDescribeCol SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY007 HY010	Function sequence error.	The function is called prior to calling SQLPrepare() or SQLExecDirect() for the <i>hstmt</i> . The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The length specified in argument <i>cbColNameMax</i> is less than 1.
HYC00	Driver not capable.	The SQL data type of column <i>icol</i> is not recognized by DB2 ODBC.
S1002	Invalid column number.	The value specified for the argument <i>icol</i> is less than 1. The value specified for the argument <i>icol</i> is greater than the number of columns in the result set.

Restrictions

The ODBC defined data type SQL_BIGINT is not supported.

Example

```

/* ... */
/*****
** process_stmt
** - allocates a statement handle
** - executes the statement
** - determines the type of statement
** - if there are no result columns, therefore non-select statement
**   - if rowcount > 0, assume statement was UPDATE, INSERT, DELETE
**   else
**     - assume a DDL, or Grant/Revoke statement
**   else
**     - must be a select statement.
**     - display results
** - frees the statement 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);

```

SQLDescribeCol

```
/* execute the SQL statement in "sqlstr" */
rc = SQLExecDirect(hstmt, sqlstr, SQL_NTS);
if (rc != SQL_SUCCESS)
    if (rc == SQL_NO_DATA_FOUND) {
        printf("\nStatement executed without error, however,\n");
        printf("no data was found or modified\n");
        return (SQL_SUCCESS);
    } else
        CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc);

rc = SQLNumResultCols(hstmt, &nresultcols);

/* determine statement type */
if (nresultcols == 0) { /* statement is not a select statement */
    rc = SQLRowCount(hstmt, &rowcount);
    if (rowcount > 0) { /* assume statement is UPDATE, INSERT, DELETE */
        printf("Statement executed, %ld rows affected\n", rowcount);
    } else { /* assume statement is GRANT, REVOKE or a DLL
        * statement */
        printf("Statement completed successful\n");
    }
} else { /* display the result set */
    display_results(hstmt, nresultcols);
} /* end determine statement type */

rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt); /* free statement handle */

return (0);
} /* end process_stmt */

/*****
** display_results
**
** - for each column
**   - get column name
**   - bind column
** - display column headings
** - fetch each row
**   - if value truncated, build error message
**   - if column null, set value to "NULL"
**   - display row
**   - print truncation message
** - free local storage
*****/
display_results(SQLHSTMT hstmt,
                SQLSMALLINT nresultcols)
{
    SQLCHAR          colname[32];
    SQLSMALLINT      coltype;
    SQLSMALLINT      colnamelen;
    SQLSMALLINT      nullable;
    SQLINTEGER        collen[MAXCOLS];
    SQLUINTEGER       precision;
    SQLSMALLINT       scale;
    SQLINTEGER        outlen[MAXCOLS];
    SQLCHAR          *data[MAXCOLS];
    SQLCHAR          errmsg[256];
    SQLRETURN         rc;
    SQLINTEGER        i;
    SQLINTEGER        x;
    SQLINTEGER        displaysize;
}
```

```

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_DESC_DISPLAY_SIZE, NULL, 0,
                   NULL, &displaysize);

    /*
     * set column length to max of display length, and column name
     * length. Plus one byte for null terminator
     */
    collen[i] = max(displaysize, strlen((char *) colname)) + 1;

    printf("%-*.*s", collen[i], collen[i], colname);

    /* allocate memory to bind column */
    data[i] = (SQLCHAR *) malloc(collen[i]);

    /* bind columns to program vars, converting all types to CHAR */
    rc = SQLBindCol(hstmt, i + 1, SQL_C_CHAR, data[i], collen[i], &outlen[i]);
}
printf("\n");

/* display result rows */
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA_FOUND) {
    errmsg[0] = '\0';
    for (i = 0; i < nresultcols; i++) {
        /* Build a truncation message for any columns truncated */
        if (outlen[i] >= collen[i]) {
            sprintf((char *) errmsg + strlen((char *) errmsg),
                  "%ld chars truncated, col %d\n",
                  outlen[i] - collen[i] + 1, i + 1);
            sprintf((char *) errmsg + strlen((char *) errmsg),
                  "Bytes to return = %ld size of buffer\n",
                  outlen[i], collen[i]);
        }
        if (outlen[i] == SQL_NULL_DATA)
            printf("%-*.*s", collen[i], collen[i], "NULL");
        else
            printf("%-*.*s", collen[i], collen[i], data[i]);
    }
    /* for all columns in this row */
    printf("\n%s", errmsg); /* print any truncation messages */
}
/* while rows to fetch */

/* free data buffers */
for (i = 0; i < nresultcols; i++) {
    free(data[i]);
}

}
/* ... */
/* end display_results */

```

References

- “SQLColAttributes - Get column attributes” on page 114
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLNumResultCols - Get number of result columns” on page 294
- “SQLPrepare - Prepare a statement” on page 300

SQLDescribeParam - Describe parameter marker

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

SQLDescribeParam() retrieves the description of a parameter marker associated with a prepared statement.

Either SQLPrepare() or SQLExecDirect() must be called before calling this function.

Syntax

```
SQLRETURN SQLDescribeParam (SQLHSTMT          hstmt,
                             SQLUSMALLINT      ipar,
                             SQLSMALLINT FAR   *pfSqlType,
                             SQLUINTEGER FAR   *pcbColDef,
                             SQLSMALLINT FAR   *pibScale,
                             SQLSMALLINT FAR   *pfnNullable);
```

Function arguments

Table 46. SQLDescribeParam arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLUSMALLINT	ipar	input	Parameter marker number ordered sequentially left to right in prepared SQL statement, starting from 1.
SQLSMALLINT *	pfSqlType	output	Base SQL data type.
SQLUINTEGER *	pcbColDef	output	Precision of the parameter marker. See Appendix D, "Data conversion", on page 485 for more details on precision, scale, length, and display size.
SQLSMALLINT *	pibScale	output	Scale of the parameter marker. See Appendix D, "Data conversion", on page 485 for more details on precision, scale, length, and display size.
SQLSMALLINT *	pfnNullable	output	Indicates whether the parameter allows NULL values. Returns one of the following values: <ul style="list-style-type: none"> SQL_NO_NULLS: The parameter does not allow NULL values (this is the default). SQL_NULLABLE: The parameter allows NULL values. SQL_NULLABLE_UNKNOWN: The driver cannot determine if the parameter allows NULL values.

Usage

For distinct types, SQLDescribeParam() returns both base data types for the input parameter.

For information about a parameter marker associated with the SQL CALL statement, use the SQLProcedureColumns() function.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 47. SQLDescribeParam SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message indicating an internal commit was issued on behalf of the application as part of the processing to set the specified connection option.
07009	Invalid descriptor index.	The value specified for the argument <i>ipar</i> is less than 1 or greater than the maximum number of parameters supported by the server.
HY000	General error.	An error occurred for which there is no specific SQLSTATE and for which no specific SQLSTATE is defined. The error message returned by SQLGetDiagRec() in the argument <i>szErrorMsg</i> describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HYC00	Driver not capable.	The data source does not support the description of input parameters.

Restrictions

None.

References

- “SQLBindParameter - Binds a parameter marker to a buffer or LOB locator” on page 91
- “SQLCancel - Cancel statement” on page 102
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLExecute - Execute a statement” on page 166
- “SQLPrepare - Prepare a statement” on page 300

SQLDisconnect - Disconnect from a data source

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLDisconnect() closes the connection associated with the database connection handle.

SQLEndTran() must be called before calling SQLDisconnect() if an outstanding transaction exists on this connection.

After calling this function, either call SQLConnect() to connect to another database, or call SQLFreeHandle().

Syntax

```
SQLRETURN SQLDisconnect (SQLHDBC hdbc);
```

Function arguments

Table 48. SQLDisconnect arguments

Data type	Argument	Use	Description
SQLHDBC	<i>hdbc</i>	input	Connection handle

Usage

If an application calls SQLDisconnect() before it has freed 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, a problem was encountered on the clean up subsequent to the disconnect, or if there is no current connection because of an event that occurred independently of the application (such as communication failure).

After a successful SQLDisconnect() call, the application can re-use *hdbc* to make another SQLConnect() or SQLDriverConnect() request.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 49. SQLDisconnect SQLSTATEs

SQLSTATE	Description	Explanation
01002	Disconnect error.	An error occurred during the disconnect. However, the disconnect succeeded. (Function returns SQL_SUCCESS_WITH_INFO.)

Table 49. SQLDisconnect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
08003	Connection is closed.	The connection specified in the argument <i>hdbc</i> is not open.
25000 25501	Invalid transaction state.	A transaction is in process on the connection specified by the argument <i>hdbc</i> . The transaction remains active, and the connection cannot be disconnected. Note: This error does not apply to stored procedures written in DB2 ODBC.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

See “Example” on page 74.

References

- “SQLAllocHandle - Allocate handle” on page 79
- “SQLConnect - Connect to a data source” on page 129
- “SQLDriverConnect - (Expanded) connect to a data source” on page 146
- “SQLTransact - Transaction management” on page 386

SQLDriverConnect - (Expanded) connect to a data source

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLDriverConnect() is an alternative to SQLConnect(). Both functions establish a connection to the target database, but SQLDriverConnect() supports additional connection parameters.

Use SQLDriverConnect() when you want to pass any or all keyword values defined in the DB2 ODBC initialization file.

When a connection is established, the completed connection string is returned. Applications can store this string for future connection requests. This allows you to override any or all keyword values in the DB2 ODBC initialization file.

Syntax

Generic

```
SQLRETURN SQLDriverConnect (SQLHDBC          hdbc,
                             SQLHWND         hwnd,
                             SQLCHAR          FAR *szConnStrIn,
                             SQLSMALLINT     cbConnStrIn,
                             SQLCHAR          FAR *szConnStrOut,
                             SQLSMALLINT     cbConnStrOutMax,
                             SQLSMALLINT     FAR *pcbConnStrOut,
                             SQLUSMALLINT    fDriverCompletion);
```

Function arguments

Table 50. SQLDriverConnect arguments

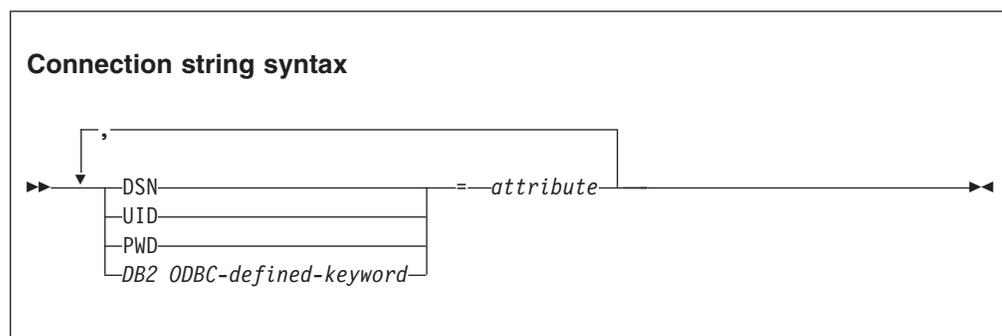
Data type	Argument	Use	Description
SQLHDBC	<i>hdbc</i>	input	Connection handle.
SQLHWND	<i>hwnd</i>	input	NULL value. Not used.
SQLCHAR *	<i>szConnStrIn</i>	input	A full, partial or empty (null pointer) connection string (see syntax and description below).
SQLSMALLINT	<i>cbConnStrIn</i>	input	Length of <i>szConnStrIn</i> .
SQLCHAR *	<i>szConnStrOut</i>	output	Pointer to buffer for the completed connection string. 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	<i>cbConnStrOutMax</i>	input	Maximum size of the buffer pointed to by <i>szConnStrOut</i> .
SQLCHAR *	<i>pcbConnStrOut</i>	output	Pointer to the number of bytes available to return in the <i>szConnStrOut</i> buffer. If the value of <i>pcbConnStrOut</i> is greater than or equal to <i>cbConnStrOutMax</i> , the completed connection string in <i>szConnStrOut</i> is truncated to <i>cbConnStrOutMax</i> - 1 bytes.

Table 50. SQLDriverConnect arguments (continued)

Data type	Argument	Use	Description
SQLUSMALLINT	<i>fDriverCompletion</i>	input	Indicates when DB2 ODBC should prompt the user for more information. Possible values: <ul style="list-style-type: none"> SQL_DRIVER_PROMPT SQL_DRIVER_COMPLETE SQL_DRIVER_COMPLETE_REQUIRED SQL_DRIVER_NOPROMPT However, DB2 for OS/390 and z/OS supports SQL_DRIVER_NOPROMPT only.

Usage

The connection string is used to pass one or more values needed to complete a connection.



Each keyword above has an attribute that is equal to the following:

DSN Data source name. The name or alias-name of the database. Required if *fDriverCompletion* is equal to SQL_DRIVER_NOPROMPT.

UID Authorization-name (user identifier). This value is validated and authenticated. If there is no user ID, an empty string is specified (UID=;).

PWD The password corresponding to the authorization name. If there is no password for the user ID, an empty string is specified (PWD=;). This value is validated and authenticated.

The user is authenticated when values are specified for both UID and PWD on the connection string. When authentication is performed, a CONNECT statement is run. The CONNECT statement cannot run if the user ID is null or empty.

The list of DB2 ODBC defined keywords and their associated attribute values are discussed in "Initialization keywords" on page 55. Any one of the keywords in that section can be specified on the connection string. If any keywords are repeated in the connection string, the value associated with the first occurrence of the keyword is used.

If any keywords exist in the DB2 ODBC initialization file, the keywords and their respective values are used to augment the information passed to DB2 ODBC in the

SQLDriverConnect

connection string. If the information in the DB2 ODBC initialization file contradicts information in the connection string, the values in 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 contained in the connection string. If there is not enough information, SQL_ERROR is returned.

When a connection is established, the complete connection string is returned.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

All of the diagnostics generated by “SQLConnect - Connect to a data source” on page 129 can be returned here as well. The following table shows the additional diagnostics that can be returned.

Table 51. SQLDriverConnect SQLSTATES

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer <i>szConnstrOut</i> is not large enough to hold the entire connection string. The argument <i>pcbConnStrOut</i> contains the actual length of the connection string available for return. (Function returns SQL_SUCCESS_WITH_INFO)
01S00	Invalid connection string attribute.	An invalid keyword or attribute value is specified in the input connection string, but the connection to the data source is successful because one of the following occurred: <ul style="list-style-type: none">• The unrecognized keyword is ignored.• The invalid attribute value is ignored, the default value is used instead. (Function returns SQL_SUCCESS_WITH_INFO)
01S02	Option value changed.	SQL_CONNECTTYPE changed to SQL_CONCURRENT_TRANS when MULTICONTEXT=1 in use.
HY090	Invalid string or buffer length.	The value specified for <i>cbConnStrIn</i> is less than 0, but not equal to SQL_NTS. The value specified for <i>cbConnStrOutMax</i> is less than 0.
HY110	Invalid driver completion.	The value specified for the argument <i>fCompletion</i> is not equal to one of the valid values.

Restrictions

See restrictions described above for *fDriverCompletion* and *SQLHWND* parameters.

Example

```

/*****
/* 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;optimizeforrows=30";

    char         ConnStrOut [200];
    SQLSMALLINT  cbConnStrOut;
    int          i;
    char         *token;

    (void) printf ("**** Entering CLIP10.\n\n");

/*****
/* CONNECT to DB2 for OS/390 */
*****/

    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 = %d.\n", RETCODE);
    goto dberror;
}

/*****
/* Enumerate keywords and values returned from SQLDriverConnect */
*****/

(void) printf ("**** ConnStrOut = %s.\n", ConnStrOut);

for (i = 1, token = strtok (ConnStrOut, ";");
     (token != NULL);
     token = strtok (NULL, ";"), i++)
    (void) printf ("**** Keyword # %d is: %s.\n", i, token);

/*****
/* DISCONNECT from data source */
*****/

RETCODE = SQLDisconnect(hDbc);

if (RETCODE != SQL_SUCCESS)
    goto dberror;

/*****
/* Deallocate Connection Handle */
*****/

RETCODE = SQLFreeHandle (SQL_HANDLE_DBC, hDbc);

if (RETCODE != SQL_SUCCESS)
    goto dberror;

/*****
/* Disconnect from data sources in Connection Table */
*****/

SQLFreeHandle (SQL_HANDLE_ENV, hEnv); /* free the environment handle */

goto exit;

dberror:
RETCODE=12;

exit:

(void) printf ("**** Exiting CLIP10.\n\n");

return(RETCODE);
}

```

References

- “SQLAllocHandle - Allocate handle” on page 79
- “SQLConnect - Connect to a data source” on page 129

SQLEndTran - End transaction of a connection

Purpose

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLEndTran() requests a commit or rollback operation for all active operations on all statements associated with a connection. SQLEndTran() can also request that a commit or rollback operation be performed for all connections associated with an environment.

Syntax

```
SQLRETURN SQLEndTran (SQLSMALLINT HandleType,
                     SQLHANDLE Handle,
                     SQLSMALLINT CompletionType);
```

Function arguments

Table 52. SQLEndTran arguments

Data type	Argument	Use	Description
SQLSMALLINT	<i>HandleType</i>	input	Handle type identifier. Contains either SQL_HANDLE_ENV if <i>Handle</i> is an environment handle or SQL_HANDLE_DBC if <i>Handle</i> is a connection handle.
SQLHANDLE	<i>Handle</i>	input	The handle, of the type indicated by <i>HandleType</i> , that indicates the scope of the transaction. See "Usage" for more information.
SQLSMALLINT	<i>CompletionType</i>	input	One of the following values: <ul style="list-style-type: none"> SQL_COMMIT SQL_ROLLBACK

Usage

A new transaction is implicitly started when an SQL statement that can be contained within a transaction is executed against the current data source. The application might need to commit or rollback based on execution status.

If *HandleType* is SQL_HANDLE_ENV and *Handle* is a valid environment handle, DB2 ODBC attempts to commit or roll back transactions one at a time, depending on the value of *CompletionType*, on all connections that are in a connected state on that environment. 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 connection(s) failed during the commit or rollback operation, the application can call SQLGetDiagRec() for each connection.

If *CompletionType* is 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.

It is important to note that unless the connection option SQL_CONNECTTYPE is set to SQL_COORDINATED_TRANS (to indicate coordinated distributed

transactions), there is no attempt to provide a coordinated global transaction with one-phase or two-phase commit protocols.

Completing a transaction has the following effects:

- Prepared SQL statements (using SQLPrepare()) survive transactions; they can be executed again without first calling SQLPrepare().
- Cursor positions are maintained after a commit unless one or more of the following is true:
 - The server is DB2 Server for VSE and VM.
 - The SQL_CURSOR_HOLD statement option 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 has not been overridden by resetting the SQL_CURSOR_HOLD statement option.
 - The CURSORHOLD keyword is present in a the connection string on the SQLDriverConnect() call that set up this connection, and it indicates cursor with hold is not in effect, and this has not been overridden by resetting the SQL_CURSOR_HOLD statement option.

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, an application calls SQLGetInfo() with the SQL_CURSOR_ROLLBACK_BEHAVIOR and SQL_CURSOR_COMMIT_BEHAVIOR options.

- Cursors are closed after a rollback and all pending results are discarded.
- Statement handles are still valid after a call to SQLEndTran(), and 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.

Whether DB2 ODBC is in auto-commit mode (using the ODBC default SQL_ATTR_AUTOCOMMIT=SQL_AUTOCOMMIT_ON) or manual-commit mode (when the application calls SQLSetConnectAttr() with the SQL_ATTR_AUTOCOMMIT attribute set to SQL_AUTOCOMMIT_OFF), calling SQLEndTran() always flows the request to the database for execution.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

Table 53. SQLEndTran SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)

SQLEndTran

Table 53. SQLEndTran SQLSTATEs (continued)

SQLSTATE	Description	Explanation
08003	Connection is closed.	The <i>ConnectionHandle</i> was not in a connected state.
08007	Connection failure during transaction.	The connection associated with the <i>ConnectionHandle</i> failed during the execution of the function and it cannot be determined whether the requested COMMIT or ROLLBACK occurred before the failure.
40001	Transaction rollback.	The transaction was rolled back due to a resource deadlock with another transaction.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by SQLGetDiagRec() in the * <i>MessageText</i> buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC was not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() was called for the <i>StatementHandle</i> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY012	Invalid transaction code.	The value specified for the argument <i>CompletionType</i> was neither SQL_COMMIT nor SQL_ROLLBACK.
HY092	Option type out of range.	The value specified for the argument <i>HandleType</i> was not 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 sample program DSN8O3VP in DSN710.SDSNSAMP.

References

- “SQLGetInfo - Get general information” on page 234
- “SQLFreeHandle - Free handle resources” on page 193
- “SQLFreeStmt - Free (or reset) a statement handle” on page 196

SQLError - Retrieve error information

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

In ODBC 3.0, SQLGetDiagRec() replaces the ODBC 2.0 function SQLError(). See SQLGetDiagRec() for more information.

SQLError() returns the diagnostic information (both errors and warnings) associated with the most recently invoked DB2 ODBC function for a particular statement, connection or environment handle.

The information consists of a standardized SQLSTATE and native error code. See “Diagnostics” on page 28 for more information.

Call SQLError() after receiving a return code of SQL_ERROR or SQL_SUCCESS_WITH_INFO from another function call.

Note: Some database servers provide product-specific diagnostic information after returning SQL_NO_DATA_FOUND from the execution of a statement.

Syntax

```
SQLRETURN SQLError(
    (SQLHENV
    SQLHDBC
    SQLHSTMT
    SQLCHAR FAR
    SQLINTEGER FAR
    SQLCHAR FAR
    SQLSMALLINT
    SQLSMALLINT FAR
    henv,
    hdbc,
    hstmt,
    *szSqlState,
    *pfNativeError,
    *szErrorMsg,
    cbErrorMsgMax,
    *pcbErrorMsg);
```

Function arguments

Table 54. SQLError arguments

Data type	Argument	Use	Description
SQLHENV	<i>henv</i>	input	Environment handle. To obtain diagnostic information associated with an environment, pass a valid environment handle. Set <i>hdbc</i> and <i>hstmt</i> to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
SQLHDBC	<i>hdbc</i>	input	Database connection handle. To obtain diagnostic information associated with a connection, pass a valid database connection handle, and set <i>hstmt</i> to SQL_NULL_HSTMT. The <i>henv</i> argument is ignored.
SQLHSTMT	<i>hstmt</i>	input	Statement handle. To obtain diagnostic information associated with a statement, pass a valid statement handle. The <i>henv</i> and <i>hdbc</i> arguments are ignored.
SQLCHAR *	<i>szSqlState</i>	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.

SQLError

Table 54. *SQLError* arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER *	<i>pfNativeError</i>	output	Native error code. In DB2 ODBC, the <i>pfNativeError</i> argument contains the SQLCODE value returned by the DBMS. If the error is generated by DB2 ODBC and not the DBMS, then this field is set to -99999.
SQLCHAR *	<i>szErrorMsg</i>	output	<p>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:</p> <p>[DB2 for OS/390 and z/OS][CLI Driver]</p> <p>to indicate that it is DB2 ODBC that detected the error and there is no database connection yet.</p> <p>The error location, ERRLOC x:y:z, keyword value is embedded in the buffer also. This is an internal error code for diagnostics.</p> <p>If the error is detected while there is a database connection, then the error message returned from the DBMS is prefaced by:</p> <p>[DB2 for OS/390 and z/OS][CLI Driver][DBMS-name]</p> <p>where DBMS-name is the name returned by SQLGetInfo() with SQL_DBMS_NAME information type.</p> <p>For example,</p> <p>DB2 DB2/6000 Vendor</p> <p>Vendor indicates a non-IBM DRDA DBMS.</p> <p>If the error is generated by the DBMS, the IBM-defined SQLSTATE is appended to the text string.</p>
SQLSMALLINT	<i>cbErrorMsgMax</i>	input	The maximum (that is, the allocated) length of the buffer <i>szErrorMsg</i> . The recommended length to allocate is SQL_MAX_MESSAGE_LENGTH + 1.
SQLSMALLINT *	<i>pcbErrorMsg</i>	output	Pointer to total number of bytes available to return to the <i>szErrorMsg</i> buffer. This does not include the null termination character.

Usage

The SQLSTATEs are those defined by the X/OPEN SQL CAE and the X/Open SQL CLI CAE, augmented with IBM specific and product specific SQLSTATE values.

To obtain diagnostic information associated with:

- An environment, pass a valid environment handle. Set *hdbc* and *hstmt* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
- A connection, pass a valid database connection handle, and set *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored.
- A statement, pass a valid statement handle. The *henv* and *hdbc* arguments are ignored.

If diagnostic information generated by one DB2 ODBC function is not retrieved before a function other than SQLError() is called with the same handle, the

information for the previous function call is lost. This is true whether or not diagnostic information is generated for the second DB2 ODBC function call.

Multiple diagnostic messages might be available after a given DB2 ODBC function call. These messages can be retrieved one at a time by repeatedly calling `SQL_Error()`. For each message retrieved, `SQL_Error()` returns `SQL_SUCCESS` and removes it from the list of messages available. When there are no more messages to retrieve, `SQL_NO_DATA_FOUND` is returned, the `SQLSTATE` is set to "00000", `pfNativeError` is set to 0, and `pcbErrorMsg` and `szErrorMsg` are undefined.

Diagnostic information stored under a given handle is cleared when a call is made to `SQL_Error()` with that handle, or when another DB2 ODBC function call is made with that handle. However, information associated with a given handle type is not cleared by a call to `SQL_Error()` with an associated but different handle type: for example, a call to `SQL_Error()` with a connection handle input does not clear errors associated with any statement handles under that connection.

`SQL_SUCCESS` is returned even if the buffer for the error message (`szErrorMsg`) is too short since the application is not able to retrieve the same error message by calling `SQL_Error()` again. The actual length of the message text is returned in the `pcbErrorMsg`.

To avoid truncation of the error message, declare a buffer length of `SQL_MAX_MESSAGE_LENGTH + 1`. The message text is never longer than this.

Return codes

- `SQL_SUCCESS`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`
- `SQL_NO_DATA_FOUND`

`SQL_NO_DATA_FOUND` is returned if no diagnostic information is available for the input handle, or if all of the messages are retrieved by calls to `SQL_Error()`.

Diagnostics

`SQLSTATE`s are not defined, since `SQL_Error()` does not generate diagnostic information for itself.

Restrictions

Although ODBC also returns X/Open SQL CAE `SQLSTATE`s, only DB2 ODBC (and the DB2 ODBC driver) returns the additional IBM-defined `SQLSTATE`s. For more information on ODBC specific `SQLSTATE`s see *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*.

Because of this, you should only build dependencies on the standard `SQLSTATE`s. This means any branching logic in the application should only rely on the standard `SQLSTATE`s. The augmented `SQLSTATE`s are most useful for debugging purposes.

Note: It might be useful to build dependencies on the class (the first 2 characters) of the `SQLSTATE`s.

Example

This example shows several utility functions used by most of the other DB2 ODBC examples.

SQLError

```
/* ... */
/*****
** - print_error - call SQLError(), display SQLSTATE and message
**               - called by check_error, see below
*****/

SQLRETURN
print_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc, /* Return code to be included with error msg */
            SQLINTEGER line, /* Used for output message, indicate where */
            SQLCHAR * file) /* the error was reported from */
{
    SQLCHAR      buffer[SQL_MAX_MESSAGE_LENGTH + 1];
    SQLCHAR      sqlstate[SQL_SQLSTATE_SIZE + 1];
    SQLINTEGER   sqlcode;
    SQLSMALLINT  length;

    printf(">--- ERROR -- RC= %ld Reported from %s, line %ld -----\n",
           frc, file, line);
    while (SQLError(henv, hdbc, hstmt, sqlstate, &sqlcode, buffer,
                   SQL_MAX_MESSAGE_LENGTH + 1, &length) == SQL_SUCCESS) {
        printf("      SQLSTATE: %s\n", sqlstate);
        printf("Native Error Code: %ld\n", sqlcode);
        printf("%s \n", buffer);
    };
    printf(">-----\n");
    return (SQL_ERROR);
}

/*****
** - check_error - call print_error(), checks severity of return code
*****/
SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc,
            SQLINTEGER line,
            SQLCHAR * file)
{
    SQLRETURN    rc;

    print_error(henv, hdbc, hstmt, frc, line, file);

    switch (frc) {
    case SQL_SUCCESS:
        break;
    case SQL_INVALID_HANDLE:
        printf("\n>----- ERROR Invalid Handle -----\n");
    }
}
```

```

case SQL_ERROR:
    printf("\n>--- FATAL ERROR, Attempting to rollback transaction --\n");
    rc = SQLTransact(henv, hdbc, SQL_ROLLBACK);
    if (rc != SQL_SUCCESS)
        printf(">Rollback Failed, Exiting application\n");
    else
        printf(">Rollback Successful, Exiting application\n");
    exit(0);
    break;
case SQL_SUCCESS_WITH_INFO:
    printf("\n> ----- Warning Message, application continuing ----- \n");
    break;
case SQL_NO_DATA_FOUND:
    printf("\n> ----- No Data Found, application continuing ----- \n");
    break;
default:
    printf("\n> ----- Invalid Return Code ----- \n");
    printf("> ----- Attempting to rollback transaction ----- \n");
    SQLTransact(henv, hdbc, SQL_ROLLBACK);
    exit(0 );
    break;
}
return (SQL_SUCCESS);

}
/* ... */

}
/*****
* The following macros use check_error
*
* {check_error(henv, SQL_NULL_HDBC, SQL_NULL_HSTMT, RC, __LINE__, __FILE__);}
*
* {check_error(SQL_NULL_HENV, hdbc, SQL_NULL_HSTMT, RC, __LINE__, __FILE__);}
*
* {check_error(SQL_NULL_HENV, SQL_NULL_HDBC, hstmt, RC, __LINE__, __FILE__);}
*
*****/

/*****
** - check_error - call print_error(), checks severity of return code
*****/
SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc,
            SQLINTEGER line,
            SQLCHAR * file)
{
    SQLRETURN rc;

    print_error(henv, hdbc, hstmt, frc, line, file);

```

SQLError

```
switch (frc) {
case SQL_SUCCESS:
    break;
case SQL_INVALID_HANDLE:
    printf("\n>----- ERROR Invalid Handle ----- \n");
case SQL_ERROR:
    printf("\n>--- FATAL ERROR, Attempting to rollback transaction -- \n");
    rc = SQLTransact(henv, hdbc, SQL_ROLLBACK);
    if (rc != SQL_SUCCESS)
        printf(">Rollback Failed, Exiting application\n");
    else
        printf(">Rollback Successful, Exiting application\n");
    exit(terminate(henv, frc));
    break;
case SQL_SUCCESS_WITH_INFO:
    printf("\n> ----- Warning Message, application continuing ----- \n");
    break;
case SQL_NO_DATA_FOUND:
    printf("\n> ----- No Data Found, application continuing ----- \n");
    break;
default:
    printf("\n> ----- Invalid Return Code ----- \n");
    printf("> ----- Attempting to rollback transaction ----- \n");
    SQLTransact(henv, hdbc, SQL_ROLLBACK);
    exit(terminate(henv, frc));
    break;
}
return (SQL_SUCCESS);
}
/* ... */
/* end check_error */
```

References

- “SQLGetSQLCA - Get SQLCA data structure” on page 263

SQLExecDirect - Execute a statement directly

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLExecDirect() directly executes the specified SQL statement. The statement can only be executed once. Also, the connected database server must be able to dynamically prepare statement. (For more information about supported SQL statements see Table 1 on page 10.)

Syntax

```
SQLRETURN SQLExecDirect (SQLHSTMT hstmt,
                          SQLCHAR FAR *szSqlStr,
                          SQLINTEGER cbSqlStr);
```

Function arguments

Table 55. SQLExecDirect arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle. There must not be an open cursor associated with <i>hstmt</i> , see “SQLFreeStmt - Free (or reset) a statement handle” on page 196 for more information.
SQLCHAR *	<i>szSqlStr</i>	input	SQL statement string. The connected database server must be able to prepare the statement, see Table 1 on page 10 for more information.
SQLINTEGER	<i>cbSqlStr</i>	input	Length of contents of <i>szSqlStr</i> argument. The length must be set to either the exact length of the statement, or if the statement is null-terminated, set to SQL_NTS.

Usage

If the SQL statement text contains vendor escape clause sequences, DB2 ODBC first modifies the SQL statement text to the appropriate DB2-specific format before submitting it for preparation and execution. If the application does not generate SQL statements that contain vendor escape clause sequences (“Using vendor escape clauses” on page 448), then it should set the SQL_NO_SCAN statement option to SQL_NOSCAN_ON at the connection level so that each statement passed to DB2 ODBC does not incur the performance impact of scanning for vendor escape clauses.

The SQL statement cannot be a COMMIT or ROLLBACK. Instead, SQLEndTran() must be called to issue COMMIT or ROLLBACK. For more information about supported SQL statements see Table 1 on page 10.

The SQL statement string can contain parameter markers. A parameter marker is represented by a “?” character, and is used to indicate a position in the statement where an application supplied value is to be substituted when SQLExecDirect() is called. This value can be obtained from:

- An application variable.
SQLSetParam() or SQLBindParameter() is used to bind the application storage area to the parameter marker.
- A LOB value residing at the server referenced by a LOB locator.

SQLExecDirect

SQLBindParameter() or SQLSetParam() is used to bind a LOB locator to a parameter marker. The actual value of the LOB is kept at the server and does not need to be transferred to the application before being used as the input parameter value for another SQL statement.

All parameters must be bound before calling SQLExecDirect().

If the SQL statement is a query, SQLExecDirect() generates a cursor name, and open the 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.

There must not already be an open cursor on the statement handle.

If SQLParamOptions() is called to specify that an array of input parameter values is bound to each parameter marker, then the application needs to call SQLExecDirect() only once to process the entire array of input parameter values.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA
- SQL_NO_DATA_FOUND

SQL_NEED_DATA is returned when the application requests to input data-at-execution parameter values by calling SQLParamData() and SQLPutData().

SQL_SUCCESS is returned if the SQL statement is a searched UPDATE or searched DELETE and no rows satisfy the search condition. Use SQLRowCount() to determine the number of rows in a table that were affected by an UPDATE, INSERT, or DELETE statement executed against the table, or a view of the table.

Diagnostics

Table 56. SQLExecDirect SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	szSqlStr contains an UPDATE or DELETE statement but no WHERE clause. (Function returns SQL_SUCCESS_WITH_INFO or SQL_NO_DATA_FOUND if there are no rows in the table).
07001	Wrong number of parameters.	The number of parameters bound to application variables using SQLBindParameter() is less than the number of parameter markers in the SQL statement contained in the argument szSqlStr.
07006	Invalid conversion.	Transfer of data between DB2 ODBC and the application variables would result in incompatible data conversion.

Table 56. SQLExecDirect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
21S01	Insert value list does not match column list.	<i>szSqlStr</i> contains an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	<i>szSqlStr</i> 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.
22001	String data right truncation.	A character string assigned to a character type column exceeded the maximum length of the column.
22008	Invalid datetime format or datetime field overflow.	<i>szSqlStr</i> contains an SQL statement with an invalid datetime format; that is, an invalid string representation or value was specified, or the value was an invalid date. Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	<i>szSqlStr</i> contains an SQL statement with an arithmetic expression that caused division by zero.
22018	Error in assignment.	<i>szSqlStr</i> contains an SQL statement with a parameter or literal and the value or LOB locator was incompatible with the data type of the associated table column. The length associated with a parameter value (the contents of the <i>pcbValue</i> buffer specified on <code>SQLBindParameter()</code>) is not valid. The argument <i>fSQLType</i> used in <code>SQLBindParameter()</code> or <code>SQLSetParam()</code> , denoted an SQL graphic data type, but the deferred length argument (<i>pcbValue</i>) contains an odd length value. The length value must be even for graphic data types.
23000	Integrity constraint violation.	The execution of the SQL statement is not permitted because the execution would cause integrity constraint violation in the DBMS.
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.	Results were pending on the <i>hstmt</i> from a previous query or a cursor associated with the <i>hstmt</i> had not been closed.
34000	Invalid cursor name.	<i>szSqlStr</i> contains a positioned DELETE or a positioned UPDATE and the cursor referenced by the statement being executed was not open.
37xxx ^a	Invalid SQL syntax.	<i>szSqlStr</i> contains one or more of the following: <ul style="list-style-type: none"> • A COMMIT • A ROLLBACK • An SQL statement that the connected database server could not prepare • A statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belongs is rolled back due to a deadlock or timeout.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.

SQLExecDirect

Table 56. *SQLExecDirect SQLSTATES (continued)*

SQLSTATE	Description	Explanation
42xxx	Syntax error or access rule violation	425xx indicates the authorization ID does not have permission to execute the SQL statement contained in <i>szSqlStr</i> . Other 42xxx SQLSTATES indicate a variety of syntax or access problems with the statement.
42895	The value of a host variable in the EXECUTE or OPEN statement cannot be used because of its data type	The LOB locator type specified on the bind parameter function call does not match the LOB data type of the parameter marker. The argument <i>fSQLType</i> used on the bind parameter function specified a LOB locator type but the corresponding parameter marker is not a LOB.
42S01	Database object already exists.	<i>szSqlStr</i> contains a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already exists.
42S02	Database object does not exist.	<i>szSqlStr</i> contains an SQL statement that references a table name or view name which does not exist.
42S11	Index already exists.	<i>szSqlStr</i> contains a CREATE INDEX statement and the specified index name already exists.
42S12	Index not found.	<i>szSqlStr</i> contains a DROP INDEX statement and the specified index name does not exist.
42S21	Column already exists.	<i>szSqlStr</i> contains an ALTER TABLE statement and the column specified in the ADD clause is not unique or identifies an existing column in the base table.
42S22	Column not found.	<i>szSqlStr</i> contains an SQL statement that references a column name that does not exist.
44000	Integrity constraint violation.	<i>szSqlStr</i> contains an SQL statement with a parameter or literal. This parameter value is NULL for a column defined as NOT NULL in the associated table column, or a duplicate value is supplied for a column constrained to contain only unique values, or some other integrity constraint is violated.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	<i>szSqlStr</i> is a null pointer.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY019	Numeric value out of range.	A numeric value assigned to a numeric type column caused truncation of the whole part of the number, either at the time of assignment or in computing an intermediate result. <i>szSqlStr</i> contains an SQL statement with an arithmetic expression which caused division by zero.
HY090	Invalid string or buffer length.	The argument <i>cbSqlStr</i> is less than 1 but not equal to SQL_NTS.

Note:

^a xxx refers to any SQLSTATE with that class code. Example, **37xxx** refers to any SQLSTATE in the **37** class.

Restrictions

None.

Example

See “Example” on page 180.

References

- “SQLBindParameter - Binds a parameter marker to a buffer or LOB locator” on page 91
- “SQLExecute - Execute a statement” on page 166
- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169
- “SQLFetch - Fetch next row” on page 176
- “SQLParamData - Get next parameter for which a data value is needed” on page 296
- “SQLPutData - Passing data value for a parameter” on page 327
- “SQLSetParam - Binds a parameter marker to a buffer” on page 354

SQLExecute - Execute a statement

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLExecute() executes a statement, that is successfully prepared using SQLPrepare(), once or multiple times. The statement is executed using the current value of any application variables that are bound to parameter markers by SQLBindParameter() or SQLSetParam() .

Syntax

```
SQLRETURN SQLExecute (SQLHSTMT hstmt);
```

Function arguments

Table 57. SQLExecute arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle. There must not be an open cursor associated with hstmt, see "SQLFreeStmt - Free (or reset) a statement handle" on page 196 for more information.

Usage

The SQL statement string can contain parameter markers. A parameter marker is represented by a "?" character, and is used to indicate a position in the statement where an application supplied value is to be substituted when SQLExecute() is called. This value can be obtained from:

- An application variable.
SQLSetParam() or SQLBindParameter() is used to bind the application storage area to the parameter marker.
- A LOB value residing at the server referenced by a LOB locator.
SQLBindParameter() or SQLSetParam() is used to bind a LOB locator to a parameter marker. The actual value of the LOB is kept at the server and does not need to be transferred to the application before being used as the input parameter value for another SQL statement.

You must bind all parameters 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.

A statement executed by SQLExecDirect() cannot be re-executed by calling SQLExecute(); SQLPrepare() must be called first.

If the prepared SQL statement is a query, SQLExecute() generates a cursor name, and open 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, the application must close the cursor by calling `SQLFreeStmt()` with the `SQL_CLOSE` option. There must not be an open cursor on the statement handle when calling `SQLExecute()`.

If a result set is generated, `SQLFetch()` or `SQLExtendedFetch()` retrieves the next row (or rows) of data into bound variables or LOB locators. Data can also be retrieved by calling `SQLGetData()` for any column that was not bound.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement must be positioned on a row at the time `SQLExecute()` is called, and must be defined on a separate statement handle under the same connection handle.

If `SQLParamOptions()` is called to specify that an array of input parameter values is bound to each parameter marker, then the application needs to call `SQLExecDirect()` only once to process the entire array of input parameter values. If the executed statement returns multiple result sets (one for each set of input parameters), then `SQLMoreResults()` should be used to advance to the next result set when processing on the current result set is complete. See “`SQLMoreResults - Determine if there are more result sets`” on page 286 for more information.

Return codes

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`
- `SQL_NEED_DATA`
- `SQL_NO_DATA_FOUND`

`SQL_NEED_DATA` is returned when the application requests to input data-at-execution parameter values by calling `SQLParamData()` and `SQLPutData()`.

`SQL_SUCCESS` is returned if the SQL statement is a searched UPDATE or searched DELETE and no rows satisfy the search condition. Use `SQLRowCount()` to determine the number of rows in a table that were affected by an UPDATE, INSERT, or DELETE statement executed against the table, or a view of the table.

Diagnostics

The `SQLSTATEs` for `SQLExecute()` include all those for `SQLExecDirect()` (see Table 56 on page 162) except for **HY009**, **HY014**, and **HY090**, and with the addition of **HY010**.

Restrictions

None.

Example

See “Example” on page 303.

References

- “`SQLBindParameter - Binds a parameter marker to a buffer or LOB locator`” on page 91
- “`SQLExecDirect - Execute a statement directly`” on page 161
- “`SQLExecute - Execute a statement`” on page 166
- “`SQLExtendedFetch - Extended fetch (fetch array of rows)`” on page 169

SQLExecute

- “SQLFetch - Fetch next row” on page 176
- “SQLParamOptions - Specify an input array for a parameter” on page 298
- “SQLPrepare - Prepare a statement” on page 300
- “SQLSetParam - Binds a parameter marker to a buffer” on page 354

SQLExtendedFetch - Extended fetch (fetch array of rows)

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLExtendedFetch() extends the function of SQLFetch() by returning a block of data containing multiple rows (called a *rowset*), in the form of an array, for each bound column. The size of the rowset is determined by the SQL_ROWSET_SIZE option on an SQLSetStmtAttr() call.

To fetch one row of data at a time, an application should call SQLFetch().

For more description on block or array retrieval, see “Retrieving a result set into an array” on page 406.

Syntax

```
SQLRETURN SQLExtendedFetch (SQLHSTMT          hstmt,
                             SQLUSMALLINT      fFetchType,
                             SQLINTEGER         irow,
                             SQLINTEGER FAR     *pcrow,
                             SQLUSMALLINT FAR   *rgfRowStatus);
```

Function arguments

Table 58. SQLExtendedFetch arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLUSMALLINT	fFetchType	Input	Direction and type of fetch. DB2 ODBC only supports the fetch direction SQL_FETCH_NEXT; that is, forward only cursor direction. The next array (rowset) of data is retrieved.
SQLINTEGER	irow	Input	Reserved for future use.
SQLINTEGER *	pcrow	Output	Number of the rows actually fetched. If an error occurs during processing, <i>pcrow</i> points to the ordinal position of the row (in the rowset) that precedes the row where the error occurred. If an error occurs retrieving the first row <i>pcrow</i> points to the value 0.
SQLUSMALLINT *	rgfRowStatus	Output	An array of status values. The number of elements must equal the number of rows in the rowset (as defined by the SQL_ROWSET_SIZE option). A status value for each row fetched is returned: <ul style="list-style-type: none"> SQL_ROW_SUCCESS <p>If the number of rows fetched is less than the number of elements in the status array (i.e. less than the rowset size), the remaining status elements are set to SQL_ROW_NOROW.</p> <p>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:</p> <ul style="list-style-type: none"> SQL_ROW_DELETED SQL_ROW_UPDATED

Usage

SQLExtendedFetch() performs an array fetch of a set of rows. An application specifies the size of the array by calling SQLSetStmtAttr() with the SQL_ROWSET_SIZE option.

Before SQLExtendedFetch() is called the first time, the cursor is positioned before the first row. After SQLExtendedFetch() is called, the cursor is positioned on the row in the result set corresponding to the last row element in the rowset just retrieved.

For any columns in the result set that are bound using the SQLBindCol() function, DB2 ODBC converts the data for the bound columns as necessary and stores it in the locations bound to these columns. As mentioned in section “Retrieving a result set into an array” on page 406, the result set can be bound in a column-wise or row-wise fashion.

- For column-wise binding of application variables:

To bind a result set in column-wise fashion, an application specifies SQL_BIND_BY_COLUMN for the SQL_BIND_TYPE statement option. (This is the default value.) Then the application calls the SQLBindCol() function.

When the application calls SQLExtendedFetch(), data for the first row is stored at the start of the buffer. Each subsequent row of data is stored at an offset of *cbValueMax* bytes (argument on SQLBindCol() call) or, if the associated C buffer type is fixed width (such as SQL_C_LONG), at an offset corresponding to that fixed length from the data for the previous row.

For each bound column, the number of bytes available to return for each element is stored in the *pcbValue* array buffer (deferred output argument on SQLBindCol()) buffer bound to the column. The number of bytes available to return for the first row of that column is stored at the start of the buffer, and the number of bytes available to return for each subsequent row is stored at an offset of *sizeof(SQLINTEGER)* bytes from the value for the previous row. If the data in the column is NULL for a particular row, the associated element in the *pcbValue* array is set to SQL_NULL_DATA.

- For row-wise binding of application variables:

The application needs to first call SQLSetStmtAttr() with the SQL_BIND_TYPE option, with the *vParam* argument set to the size of the structure capable of holding a single row of retrieved data and the associated data lengths for each column data value.

For each bound column, the first row of data is stored at the address given by the *rgbValue* supplied on the SQLBindCol() call for the column and each subsequent row of data at an offset of *vParam* bytes (used on the SQLSetStmtAttr() call) from the data for the previous row.

For each bound column, the number of bytes available to return for the first row is stored at the address given by the *pcbValue* argument supplied on the SQLBindCol() call, and the number of bytes available to return for each subsequent row at an offset of *vParam* bytes from address containing the value for the previous row.

If SQLExtendedFetch() returns an error that applies to the entire rowset, the SQL_ERROR function return code is reported with the appropriate SQLSTATE. The contents of the rowset buffer are undefined and the cursor position is unchanged.

If an error occurs that applies to a single row:

- The corresponding element in the *rgbRowStatus* array for the row is set to SQL_ROW_ERROR

- An SQLSTATE of **01S01** is added to the list of errors that can be obtained using `SQLGetDiagRec()`
- Zero or more additional SQLSTATES, describing the error for the current row, are added to the list of errors that can be obtained using `SQLGetDiagRec()`

An `SQL_ROW_ERROR` in the `rgfRowStatus` array only indicates that there was an error with the corresponding element; it does not indicate how many SQLSTATES were generated. Therefore, SQLSTATE **01S01** is used as a separator between the resulting SQLSTATES for each row. DB2 ODBC continues to fetch the remaining rows in the rowset and returns `SQL_SUCCESS_WITH_INFO` as the function return code. After `SQLExtendedFetch()` returns, for each row encountering an error there is an SQLSTATE of **01S01** and zero or more additional SQLSTATES indicating the errors for the current row, retrievable using `SQLGetDiagRec()`. Individual errors that apply to specific rows do not affect the cursor which continues to advance.

The number of elements in the `rgfRowStatus` array output buffer must equal the number of rows in the rowset (as defined by the `SQL_ROWSET_SIZE` statement option). If the number of rows fetched is less than the number of elements in the status array, the remaining status elements are set to `SQL_ROW_NOROW`.

An application cannot mix `SQLExtendedFetch()` with `SQLFetch()` calls.

Return codes

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`
- `SQL_NO_DATA_FOUND`

Diagnostics

Table 59. `SQLExtendedFetch` SQLSTATES

SQLSTATE	Description	Explanation
01004	Data truncated.	The data returned for one <i>or more</i> columns is truncated. (Function returns <code>SQL_SUCCESS_WITH_INFO</code> .)
01S01	Error in row.	An error occurred while fetching one or more rows. (Function returns <code>SQL_SUCCESS_WITH_INFO</code> .)
07002	Too many columns.	A column number specified in the binding for one or more columns is greater than the number of columns in the result set. The application has used <code>SQLSetColAttributes()</code> to inform DB2 ODBC of the descriptor information of the result set, but it did not provide this for every column in the result set.
07006	Invalid conversion.	The data value could not be converted in a meaningful manner to the data type specified by <code>fCType</code> in <code>SQLBindCol()</code> .
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument <code>pcbValue</code> in <code>SQLBindCol()</code> is a null pointer and the value of the corresponding column is null. There is no means to report <code>SQL_NULL_DATA</code> .

SQLExtendedFetch

Table 59. SQLExtendedFetch SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22008	Invalid datetime format or datetime field overflow.	<p>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.</p> <p>The value of a date, time, or timestamp does not conform to the syntax for the specified data type.</p> <p>Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.</p>
22012	Division by zero is invalid.	A value from an arithmetic expression is returned which results in division by zero.
22018	Error in assignment.	<p>A returned value is incompatible with the data type of the bound column.</p> <p>A returned LOB locator was incompatible with the data type of the bound column.</p>
24000	Invalid cursor state.	The previous SQL statement executed on the statement handle is not a query.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	<p>SQLExtendedFetch() is called for an <i>hstmt</i> after SQLFetch() is called and before SQLFreeStmt() is called with the SQL_CLOSE option.</p> <p>The function is called prior to calling SQLPrepare() or SQLExecDirect() for the <i>hstmt</i>.</p> <p>The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.</p>
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY019	Numeric value out of range.	<p>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.</p> <p>A value from an arithmetic expression is returned which results in division by zero.</p>
HY106	Fetch type out of range.	The value specified for the argument <i>fFetchType</i> is not recognized.
HYC00	Driver not capable.	<p>DB2 ODBC or the data source does not support the conversion specified by the combination of the <i>fCType</i> in SQLBindCol() and the SQL data type of the corresponding column.</p> <p>A call to SQLBindCol() is made for a column data type which is not supported by DB2 ODBC.</p> <p>The specified fetch type is recognized, but not supported.</p>

Restrictions

None.

Example

```

/* ... */
"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;
/* ... */

```

SQLExtendedFetch

```
/******  
/* 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_1);  
  
rc = SQLBindCol(hstmt, 3, SQL_C_SSHORT, (SQLPOINTER) id, 0, NULL);  
  
rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) name, 10, name_1);  
  
/* Fetch ROWSET_SIZE rows at a time, and display */  
printf("\nDEPTNUMB DEPTNAME ID NAME\n");  
printf("-----\n");  
while ((rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, Row_Stat))  
== SQL_SUCCESS) {  
    for (i = 0; i < pcrow; i++) {  
        printf("%8ld %-14s %8ld %-9s\n", deptnumb[i], deptname[i],  
            id[i], name[i]);  
    }  
    if (pcrow < ROWSET_SIZE)  
        break;  
} /* endwhile */  
  
if (rc != SQL_NO_DATA_FOUND && rc != SQL_SUCCESS)  
    CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);  
  
rc = SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
```

```

/*****
/* 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].deptnum) + 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].deptnum, 0,
    &R[0].deptnum_1);

rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) R[0].deptname, 15,
    &R[0].deptname_1);

rc = SQLBindCol(hstmt, 3, SQL_C_SSHORT, (SQLPOINTER) & R[0].id, 0,
    &R[0].id_1);

rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) R[0].name, 10, &R[0].name_1);

/* Fetch ROWSET_SIZE rows at a time, and display */
printf("\nDEPTNUMB DEPTNAME ID NAME\n");
printf("-----\n");
while ((rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, Row_Stat))
    == SQL_SUCCESS) {
    for (i = 0; i < pcrow; i++) {
        printf("%8ld %-14s %8ld %-9s\n", R[i].deptnum, 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 */
/* ... */

```

References

- “SQLExecute - Execute a statement” on page 166
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLFetch - Fetch next row” on page 176

SQLFetch - Fetch next row

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLFetch() advances the cursor to the next row of the result set, and retrieves any bound columns.

Columns can be bound to:

- Application storage
- LOB locators

When SQLFetch() is called, the appropriate data transfer is performed, along with any data conversion if conversion was indicated when the column was bound. The columns can also be received individually after the fetch, by calling SQLGetData().

SQLFetch() can only be called after a result set is generated (using the same statement handle) by either executing a query, calling SQLGetTypeInfo() or calling a catalog function.

To retrieve multiple rows at a time, use SQLExtendedFetch().

Syntax

```
SQLRETURN SQLFetch (SQLHSTMT hstmt);
```

Function arguments

Table 60. SQLFetch arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle

Usage

SQLFetch() can only be called after a result set is generated on the same statement handle. Before SQLFetch() is called the first time, the cursor is positioned before the start of the result set.

The number of application variables bound with SQLBindCol() must not exceed the number of columns in the result set or SQLFetch() fails.

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 variable length data types, binding columns provides better performance than using SQLGetData().

Columns can be bound to:

- Application storage

SQLBindCol() is used to bind application storage to the column. Data is transferred from the server to the application at fetch time. Length of the available data to return is also set.

- LOB locators

SQLBindCol() is used to bind LOB locators to the column. Only the LOB locator (4 bytes) is transferred from the server to the application at fetch time.

When an 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.

If LOB values are too large to retrieve in one fetch, they can be retrieved in pieces by either using SQLGetData() (which can be used for any column type), or by binding a LOB locator, and using SQLGetSubString().

If any bound storage 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 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 the diagnostics section).

Truncation of graphic data types is treated the same as character data types, except that the *rgbValue* buffer is filled to the nearest multiple of two bytes that is still less than or equal to the *cbValueMax* specified in SQLBindCol(). Graphic (DBCS) data transferred between DB2 ODBC and the application is not null-terminated if the C buffer type is SQL_C_CHAR. If the buffer type is SQL_C_DBCHAR, then null-termination of graphic data does occur.

Truncation is also affected by the SQL_MAX_LENGTH statement option. The application can specify that DB2 ODBC should not report truncation by calling SQLSetStmtAttr() with SQL_MAX_LENGTH and a value for the maximum length to return for any one column, and by allocating an *rgbValue* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS is returned and the maximum length, not the actual length is returned in *pcbValue*.

When all the rows are retrieved from the result set, or the remaining rows are not needed, SQLFreeStmt() or SQLCloseCursor() should be called to close the cursor and discard the remaining data and associated resources.

To retrieve multiple rows at a time, use SQLExtendedFetch(). An application cannot mix SQLFetch() with SQLExtendedFetch() calls on the same statement handle.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

SQLFetch

- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND is returned if there are no rows in the result set, or previous SQLFetch() calls have fetched all the rows from the result set.

If all the rows were fetched, the cursor is positioned after the end of the result set.

Diagnostics

Table 61. SQLFetch SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data returned for one or more columns is truncated. String values or numeric values are right truncated. (SQL_SUCCESS_WITH_INFO is returned if no error occurred.)
07002	Too many columns.	A column number specified in the binding for one or more columns is greater than the number of columns in the result set. The application used SQLSetColAttributes() to inform DB2 ODBC of the descriptor information of the result set, but it did not provide this for every column in the result set.
07006	Invalid conversion.	The data value cannot be converted in a meaningful manner to the data type specified by <i>fCType</i> in SQLBindCol()
07009	Invalid column number.	The specified column is less than 0 or greater than the number of result columns. The specified column is 0, but DB2 ODBC does not support ODBC bookmarks (<i>icol</i> = 0). SQLExtendedFetch() is called for this result set.
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument <i>pcbValue</i> in SQLBindCol() is a null pointer and the value of the corresponding column is null. There is no means to report SQL_NULL_DATA.
22008	Invalid datetime format or datetime field overflow.	Conversion from character string to datetime format is indicated, but an invalid string representation or value is specified, or the value is an invalid date. The value of a date, time, or timestamp does not conform to the syntax for the specified data type. Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	A value from an arithmetic expression is returned which results in division by zero.
22018	Error in assignment.	A returned value is incompatible with the data type of binding. A returned LOB locator is incompatible with the data type of the bound column.
24000	Invalid cursor state.	The previous SQL statement executed on the statement handle is not a query.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.

Table 61. SQLFetch SQLSTATES (continued)

SQLSTATE	Description	Explanation
54028	The maximum number of concurrent LOB handles has been reached.	Maximum LOB locator assigned. The maximum number of concurrent LOB locators has been reached. A new locator can not be assigned.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	SQLFetch() is called for an hstmt after SQLExtendedFetch() is called and before SQLCloseCurosr() had been called. The function is called prior to calling SQLPrepare() or SQLExecDirect() for the <i>hstmt</i> . The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY019	Numeric value out of range.	Returning the numeric value (as numeric or string) for one or more columns causes the whole part of the number to be truncated either at the time of assignment or in computing an intermediate result. A value from an arithmetic expression is returned which results in division by zero. Note: The associated cursor is undefined if this error is detected by DB2 for OS/390 and z/OS. If the error is detected by DB2 UDB or by other IBM RDBMSs, the cursor remains open and continues to advance on subsequent fetch calls.
HYC00	Driver not capable.	DB2 ODBC or the data source does not support the conversion specified by the combination of the <i>fCType</i> in SQLBindCol() and the SQL data type of the corresponding column. A call to SQLBindCol() was made for a column data type which is not supported by DB2 ODBC.

Restrictions

None.

SQLFetch

Example

```
/* ... */
/*****
** 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("%-14.14s %-14.14s \n", deptname.s, location.s);
    }
    if (rc != SQL_NO_DATA_FOUND)
        CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, RETCODE);

    rc = SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
    rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);

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

    rc = SQLDisconnect(hdbc);
    rc = SQLFreeHandle (SQL_HANDLE_DBC, hdbc);
    rc = SQLFreeHandle (SQL_HANDLE_ENV, henv);
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));
} /* end main */
/* ... */
```

References

- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169
- “SQLExecute - Execute a statement” on page 166
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLGetData - Get data from a column” on page 210

SQLForeignKeys - Get the list of foreign key columns

Purpose

Specification:	ODBC 1.0		
----------------	----------	--	--

SQLForeignKeys() returns information about foreign keys for the specified table. The information is returned in an SQL result set which can be processed using the same functions that are used to retrieve a result generated by a query.

Syntax

```
SQLRETURN SQLForeignKeys (SQLHSTMT          hstmt,
                          SQLCHAR           FAR *szPkCatalogName,
                          SQLSMALLINT      cbPkCatalogName,
                          SQLCHAR           FAR *szPkSchemaName,
                          SQLSMALLINT      cbPkSchemaName,
                          SQLCHAR           FAR *szPkTableName,
                          SQLSMALLINT      cbPkTableName,
                          SQLCHAR           FAR *szFkCatalogName,
                          SQLSMALLINT      cbFkCatalogName,
                          SQLCHAR           FAR *szFkSchemaName,
                          SQLSMALLINT      cbFkSchemaName,
                          SQLCHAR           FAR *szFkTableName,
                          SQLSMALLINT      cbFkTableName);
```

Function arguments

Table 62. SQLForeignKeys arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szPkCatalogName	input	Catalog qualifier of the primary key table. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbPkCatalogName	input	Length of <i>szPkCatalogName</i> . This must be set to 0.
SQLCHAR *	szPkSchemaName	input	Schema qualifier of the primary key table.
SQLSMALLINT	cbPkSchemaName	input	Length of <i>szPkSchemaName</i> .
SQLCHAR *	szPkTableName	input	Name of the table name containing the primary key.
SQLSMALLINT	cbPkTableName	input	Length of <i>szPkTableName</i> .
SQLCHAR *	szFkCatalogName	input	Catalog qualifier of the table containing the foreign key. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbFkCatalogName	input	Length of <i>szFkCatalogName</i> . This must be set to 0.
SQLCHAR *	szFkSchemaName	input	Schema qualifier of the table containing the foreign key.
SQLSMALLINT	cbFkSchemaName	input	Length of <i>szFkSchemaName</i> .
SQLCHAR *	szFkTableName	input	Name of the table containing the foreign key.
SQLSMALLINT	cbFkTableName	input	Length of <i>szFkTableName</i> .

Usage

If *szPkTableName* contains a table name, and *szFkTableName* 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.

SQLForeignKeys

If *szFkTableName* contains a table name, and *szPkTableName* is an empty string, `SQLForeignKeys()` returns a result set containing all of the foreign keys in the specified table and the primary keys (in other tables) to which they refer.

If both *szPkTableName* and *szFkTableName* contain table names, `SQLForeignKeys()` returns the foreign keys in the table specified in *szFkTableName* that refer to the primary key of the table specified in *szPkTableName*. This should be one key at the most.

If the schema qualifier argument associated with a table name is not specified, then the schema name defaults to the one currently in effect for the current connection.

Table 63 lists the columns of the result set generated by the `SQLForeignKeys()` call. If the foreign keys associated with a primary key are requested, the result set is ordered by `FKTABLE_CAT`, `FKTABLE_SCHEM`, `FKTABLE_NAME`, and `ORDINAL_POSITION`. If the primary keys associated with a foreign key are requested, the result set is ordered by `PKTABLE_CAT`, `PKTABLE_SCHEM`, `PKTABLE_NAME`, and `ORDINAL_POSITION`.

The `VARCHAR` columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the `SQL_MAX_CATALOG_NAME_LEN`, `SQL_MAX_SCHEMA_NAME_LEN`, `SQL_MAX_TABLE_NAME_LEN`, and `SQL_MAX_COLUMN_NAME_LEN` to determine respectively the actual lengths of the associated `TABLE_CAT`, `TABLE_SCHEM`, `TABLE_NAME`, and `COLUMN_NAME` columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 63. Columns returned by `SQLForeignKeys`

Column number/name	Data type	Description
1 PKTABLE_CAT	VARCHAR(128)	This is always NULL.
2 PKTABLE_SCHEM	VARCHAR(128)	The name of the schema containing PKTABLE_NAME.
3 PKTABLE_NAME	VARCHAR(128) NOT NULL	Name of the table containing the primary key.
4 PKCOLUMN_NAME	VARCHAR(128) NOT NULL	Primary key column name.
5 FKTABLE_CAT	VARCHAR(128)	This is always NULL.
6 FKTABLE_SCHEM	VARCHAR(128)	The name of the schema containing FKTABLE_NAME.
7 FKTABLE_NAME	VARCHAR(128) NOT NULL	The name of the table containing the foreign key.
8 FKCOLUMN_NAME	VARCHAR(128) NOT NULL	Foreign key column name.
9 KEY_SEQ	SMALLINT NOT NULL	The ordinal position of the column in the key, starting at 1.

Table 63. Columns returned by SQLForeignKeys (continued)

Column number/name	Data type	Description
10 UPDATE_RULE	SMALLINT	Action to be applied to the foreign key when the SQL operation is UPDATE: <ul style="list-style-type: none"> • SQL_RESTRICT • SQL_NO_ACTION <p>The update rule for IBM DB2 DBMSs is always either RESTRICT or SQL_NO_ACTION. However, ODBC applications might encounter the following UPDATE_RULE values when connected to non-IBM RDBMSs:</p> <ul style="list-style-type: none"> • SQL_CASCADE • SQL_SET_NULL
11 DELETE_RULE	SMALLINT	Action to be applied to the foreign key when the SQL operation is DELETE: <ul style="list-style-type: none"> • SQL_CASCADE • SQL_NO_ACTION • SQL_RESTRICT • SQL_SET_DEFAULT • SQL_SET_NULL
12 FK_NAME	VARCHAR(128)	Foreign key identifier. NULL if not applicable to the data source.
13 PK_NAME	VARCHAR(128)	Primary key identifier. NULL if not applicable to the data source.
14 DEFERRABILITY	SMALLINT	One of the following: <ul style="list-style-type: none"> • SQL_INITIALLY_DEFERRED • SQL_INITIALLY_IMMEDIATE • SQL_NOT_DEFERRABLE <p>DB2 ODBC returns NULL.</p>

Note: 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.

DB2 ODBC applications that issue SQLForeignKeys() against a DB2 for OS/390 and z/OS server, Version 5 or later, should expect the result set columns listed in the table above. Revision bars identify the new and changed columns.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 64. SQLForeignKeys SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	The arguments <i>szPkTableName</i> and <i>szFkTableName</i> are both NULL pointers.

SQLForeignKeys

Table 64. SQLForeignKeys SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal SQL_NTS. The length of the table or owner name is greater than the maximum length supported by the server. See "SQLGetInfo - Get general information" on page 234.
HYC00	Driver not capable.	DB2 ODBC does not support <i>catalog</i> as a qualifier for table name.
S1014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.

Restrictions

None.

Example

```
/* *****  
/* 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 phtable_schem;
VARCHAR phtable_name;
VARCHAR pkcolumn_name;
VARCHAR fhtable_schem;
VARCHAR fhtable_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;

```

SQLForeignKeys

```

/*****
/* Invoke SQLForeignKeys against PARENT Table, specifying NULL
/* for table with foreign key.
*****/

rc = SQLForeignKeys (hStmt,
                    NULL,
                    0,
                    (SQLCHAR *) "ADMFO01",
                    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_RESTRIC) // isolate update rule
        strcpy (update, "RESTRICT");
    else
        if (update_rule == SQL_CASCADE)
            strcpy (update, "CASCADE");
        else
            strcpy (update, "SET NULL");
}

```

SQLForeignKeys

```
    if (delete_rule == SQL_RESTRIC)    // isolate delete rule
        strcpy (delet, "RESTRIC");
    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;
}
```

References

- “SQLPrimaryKeys - Get primary key columns of a table” on page 308
- “SQLStatistics - Get index and statistics information for a base table” on page 374

SQLFreeConnect - Free connection handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

In ODBC 3.0, `SQLFreeHandle()` replaces the ODBC 2.0 function `SQLFreeConnect()`. See `SQLFreeHandle()` for more information.

`SQLFreeConnect()` invalidates and frees the connection handle. All DB2 ODBC resources associated with the connection handle are freed.

`SQLDisconnect()` must be called before calling this function.

Syntax

```
SQLRETURN SQLFreeConnect (SQLHDBC          hdbc);
```

Function arguments

Table 65. *SQLFreeConnect* arguments

Data type	Argument	Use	Description
SQLHDBC	<i>hdbc</i>	input	Connection handle

Usage

If this function is called when a connection still exists, `SQL_ERROR` is returned, and the connection handle remains valid.

To continue termination, call `SQLFreeEnv()`, or, if a new connection handle is required, call `SQLAllocConnect()`.

Return codes

- `SQL_SUCCESS`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`
- `SQL_SUCCESS_WITH_INFO`

Diagnostics

Table 66. *SQLFreeConnect* `SQLSTATE`s

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called prior to <code>SQLDisconnect()</code> for the <i>hdbc</i> .
S1013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.

Restrictions

None.

SQLFreeConnect

Example

See “Example” on page 74.

References

- “SQLDisconnect - Disconnect from a data source” on page 144
- “SQLFreeEnv - Free environment handle” on page 191

SQLFreeEnv - Free environment handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

In ODBC 3.0, `SQLFreeHandle()` replaces the ODBC 2.0 function `SQLFreeEnv()`. See `SQLFreeHandle()` for more information.

`SQLFreeEnv()` invalidates and frees the environment handle. All DB2 ODBC resources associated with the environment handle are freed.

`SQLFreeConnect()` must be called before calling this function.

This function is the last DB2 ODBC step an application needs to do before terminating.

Syntax

```
SQLRETURN SQLFreeEnv (SQLHENV henv);
```

Function arguments

Table 67. *SQLFreeEnv* arguments

Data type	Argument	Use	Description
SQLHENV	<i>henv</i>	input	Environment handle

Usage

If this function is called when there is still a valid connection handle, `SQL_ERROR` is returned, and the environment handle remains valid.

The number of `SQLFreeEnv()` calls must equal the number of `SQLAllocEnv()` calls before the environment information is reset.

Return codes

- `SQL_SUCCESS`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`

Diagnostics

Table 68. *SQLFreeEnv* `SQLSTATE`s

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	There is an <i>hdbc</i> which is in allocated or connected state. Call <code>SQLDisconnect()</code> and <code>SQLFreeConnect()</code> for the <i>hdbc</i> before calling <code>SQLFreeEnv()</code> .
S1013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.

SQLFreeEnv

Restrictions

None.

Example

See “Example” on page 74.

References

- “SQLFreeConnect - Free connection handle” on page 189

SQLFreeHandle - Free handle resources

Purpose

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLFreeHandle() frees an environment, connection, or statement handle.

SQLFreeHandle() is a generalized function for allocating handles that replaces the deprecated ODBC 2.0 functions SQLFreeEnv(), SQLFreeConnect(), and SQLFreeStmt() (with the SQL_DROP option) for freeing a statement handle.

Syntax

```
SQLRETURN SQLFreeHandle (SQLSMALLINT HandleType,
                          SQLHANDLE Handle);
```

Function arguments

Table 69. SQLFreeHandle arguments

Data type	Argument	Use	Description
SQLSMALLINT	<i>HandleType</i>	input	The type of handle to be freed by SQLFreeHandle(). Must be one of the following values: <ul style="list-style-type: none"> SQL_HANDLE_ENV SQL_HANDLE_DBC SQL_HANDLE_STMT
SQLHANDLE	<i>Handle</i>	input	Handle to be freed.

Usage

SQLFreeHandle() is used to free handles for environments, connections, and statements. An application should not use a handle after it has been freed.

- **Freeing an environment handle**

Prior to calling SQLFreeHandle() with a *HandleType* of SQL_HANDLE_ENV, an application must call SQLFreeHandle() with a *HandleType* of SQL_HANDLE_DBC for all connections allocated under the environment. Otherwise, the call to SQLFreeHandle() returns SQL_ERROR and the environment and any active connection remains valid.

- **Freeing a connection handle**

Prior to calling SQLFreeHandle() with a *HandleType* of SQL_HANDLE_DBC, an application must call SQLDisconnect() for the connection. Otherwise, the call to SQLFreeHandle() returns SQL_ERROR and the connection remains valid.

- **Freeing a statement handle**

A call to SQLFreeHandle() with a *HandleType* of SQL_HANDLE_STMT frees all resources that were allocated by a call to SQLAllocHandle() with a *HandleType* of SQL_HANDLE_STMT. When an application calls SQLFreeHandle() to free a statement that has pending results, those results are deleted.

SQLDisconnect() automatically drops any statements open on the connection.

SQLFreeHandle

Return codes

- 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

Table 70. *SQLFreeHandle* SQLSTATES

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns <code>SQL_SUCCESS_WITH_INFO</code> .)
08003	Connection is closed.	The <i>HandleType</i> argument was <code>SQL_HANDLE_DBC</code> , and the communication link between DB2 ODBC and the data source to which it was trying to connect failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by <code>SQLGetDiagRec()</code> in the <i>*MessageText</i> buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC was not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	<ul style="list-style-type: none">• The <i>HandleType</i> argument was <code>SQL_HANDLE_ENV</code>, and at least one connection was in an allocated or connected state. <code>SQLDisconnect()</code> and <code>SQLFreeHandle()</code> with a <i>HandleType</i> of <code>SQL_HANDLE_DBC</code> must be called for each connection before calling <code>SQLFreeHandle()</code> with a <i>HandleType</i> of <code>SQL_HANDLE_ENV</code>. The <i>HandleType</i> argument was <code>SQL_HANDLE_DBC</code>, and the function was called before calling <code>SQLDisconnect()</code> for the connection.• The <i>HandleType</i> argument was <code>SQL_HANDLE_STMT</code>; <code>SQLExecute()</code> or <code>SQLExecDirect()</code> was called with the statement handle, and returned <code>SQL_NEED_DATA</code>. This function was called before data was sent for all data-at-execution parameters or columns. <code>SQLCancel()</code> must be issued to free the statement handle.
HY013	Unexpected memory handling error.	The <i>HandleType</i> argument was <code>SQL_HANDLE_STMT</code> and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.

Restrictions

None.

Example

Refer to sample program `DSN8O3VP` in `DSN710.SDSNSAMP`.

References

- “`SQLAllocHandle` - Allocate handle” on page 79
- “`SQLGetDiagRec` - Get multiple field settings of diagnostic record” on page 223
- “`SQLCancel` - Cancel statement” on page 102

- “SQLDisconnect - Disconnect from a data source” on page 144

SQLFreeStmt - Free (or reset) a statement handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLFreeStmt() ends processing on the statement referenced by the statement handle. Use this function to:

- Close a cursor
- Drop the statement handle and free the DB2 ODBC resources associated with the statement handle.

SQLFreeStmt() is called after executing an SQL statement and processing the results.

Syntax

```
SQLRETURN SQLFreeStmt (SQLHSTMT hstmt,
                        SQLUSMALLINT fOption);
```

Function arguments

Table 71. SQLFreeStmt arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle
SQLUSMALLINT	<i>fOption</i>	input	Option which specified the manner of freeing the statement handle. The option must have one of the following values: <ul style="list-style-type: none"> • SQL_CLOSE • SQL_DROP • SQL_UNBIND • SQL_RESET_PARAMS

Usage

SQLFreeStmt() can be called with the following options:

SQL_CLOSE

The cursor (if any) associated with the statement handle (*hstmt*) is closed and all pending results are discarded. The application 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 *hstmt*. 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 (no warning or error is generated).

You can also call the ODBC 3.0 API SQLCloseCursor() to close the cursor. See “SQLCloseCursor - Close cursor and discard pending results” on page 104 for more information.

SQL_DROP

DB2 ODBC resources associated with the input statement handle are freed, and the handle is invalidated. The open cursor, if any, is closed and all pending results are discarded.

The SQL_DROP option is deprecated in ODBC 3.0. SQLFreeHandle() with *HandleType* set to SQL_HANDLE_STMT replaces the SQL_DROP option.

SQL_UNBIND

All the columns 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 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).

SQLFreeStmt() has no effect on LOB locators. Call SQLExecDirect() with the FREE LOCATOR statement to free a locator. See “Using large objects” on page 411 for more information on using LOBs.

You can reuse a statement handle to execute a different statement. If the handle is:

- Associated with a query, catalog function, or SQLGetTypeInfo(), you must close the cursor.
- Bound with a different number or type of parameters, the parameters must be reset.
- Bound with a different number or type of column bindings, the columns must be unbound.

Alternatively, you can drop the statement handle and allocate a new one.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

SQL_SUCCESS_WITH_INFO is not returned if *fOption* is set to SQL_DROP, since there would be no statement handle to use when SQLGetDiagRec() is called.

Diagnostics

Table 72. SQLFreeStmt SQLSTATES

SQLSTATE	Description	Explanation
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY092	Option type out of range.	The value specified for the argument <i>fOption</i> is not SQL_CLOSE, SQL_DROP, SQL_UNBIND, or SQL_RESET_PARAMS.
S1506	Error closing a file.	Error encountered while trying to close a temporary file.

Restrictions

None.

SQLFreeStmt

Example

See “Example” on page 180.

References

- “SQLAllocHandle - Allocate handle” on page 79
- “SQLBindParameter - Binds a parameter marker to a buffer or LOB locator” on page 91
- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169
- “SQLFetch - Fetch next row” on page 176
- “SQLSetParam - Binds a parameter marker to a buffer” on page 354

SQLGetConnectAttr - Get current attribute setting

Purpose

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

SQLGetConnectAttr() returns the current setting of a connection attribute. These attributes are set using the SQLSetConnectAttr() function.

Syntax

```
SQLRETURN SQLGetConnectAttr (SQLHDBC          ConnectionHandle,
                             SQLINTEGER       Attribute,
                             SQLPOINTER      ValuePtr,
                             SQLINTEGER       BufferLength,
                             SQLINTEGER       *StringLengthPtr);
```

Function arguments

Table 73. SQLGetConnectAttr arguments

Data type	Argument	Use	Description
SQLHDBC	<i>ConnectionHandle</i>	input	Connection handle.
SQLINTEGER	<i>Attribute</i>	input	Connection attribute to retrieve. Refer to Table 136 on page 337 for a complete list of attributes.
SQLPOINTER	<i>ValuePtr</i>	input	A pointer to memory in which to return the current value of the attribute specified by <i>Attribute</i> . <i>*ValuePtr</i> will be a 32-bit unsigned integer value or point to a null-terminated character string. If the <i>Attribute</i> argument is a driver-specific value, the value in <i>*ValuePtr</i> might be a signed integer.
SQLINTEGER	<i>BufferLength</i>	input	Information about the <i>*ValuePtr</i> argument. <ul style="list-style-type: none"> For ODBC-defined attributes: <ul style="list-style-type: none"> If <i>ValuePtr</i> points to a character string, this argument should be the length of <i>*ValuePtr</i>. If <i>ValuePtr</i> points to an integer, <i>BufferLength</i> is ignored. For driver-defined attributes (IBM extension): <ul style="list-style-type: none"> If <i>ValuePtr</i> points to a character string, this argument should be the length of <i>*ValuePtr</i> or SQL_NTS. If SQL_NTS, the driver assumes that the length of <i>*ValuePtr</i> is SQL_MAX_OPTIONS_STRING_LENGTH bytes (excluding the null-terminator). If <i>ValuePtr</i> points to an integer, <i>BufferLength</i> is ignored.
SQLINTEGER *	<i>StringLengthPtr</i>	output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes returned for the null-termination character) available to return in <i>ValuePtr</i> . <ul style="list-style-type: none"> If <i>ValuePtr</i> 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 <i>BufferLength</i>, the data in <i>ValuePtr</i> is truncated to <i>BufferLength</i> minus the length of a null-termination character and is null-terminated by DB2 ODBC. If <i>Attribute</i> does not denote a string, DB2 ODBC ignores <i>BufferLength</i> and does not set <i>StringLengthPtr</i>.

SQLGetConnectAttr

Usage

SQLGetConnectAttr() returns the current setting of a connection attribute. These options are set using the SQLSetConnectAttr() function. For a list of valid environment attributes, refer to Table 136 on page 337.

An application can set statement attributes using SQLSetConnectAttr(). However, an application cannot use SQLGetConnectAttr() to retrieve statement attribute values; it must call SQLGetStmtAttr() to retrieve the setting of statement attributes.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

Table 74. SQLGetConnectAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The data returned in *ValuePtr was truncated to be BufferLength minus the length of a null termination character. The length of the untruncated string value is returned in *StringLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	An Attribute value was specified that required an open connection, but the ConnectionHandle was not in a connected state.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by SQLGetDiagRec() in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC was not able to allocate memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument BufferLength was less than 0.
HY092	Option type out of range.	The value specified for the argument Attribute was not valid for this version of DB2 ODBC.
HYC00	Driver not capable.	The value specified for the argument Attribute was a valid connection or statement attribute for this version of the DB2 ODBC driver, but was not supported by the data source.

Restrictions

None.

Example

```
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" );
```

References

- “SQLSetConnectAttr - Set connection attributes” on page 336
- “SQLGetStmtAttr - Get current setting of a statement attribute” on page 270
- “SQLSetStmtAttr - Set options related to a statement” on page 360

SQLGetConnectOption - Returns current setting of a connect option

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	
-----------------------	-----------------	-------------------	--

In ODBC 3.0, SQLGetConnectAttr() replaces the ODBC 2.0 function SQLGetConnectOption(). See SQLGetConnectAttr() for more information.

SQLGetConnectOption() returns the current settings for the specified connection option.

These options are set using the SQLSetConnectOption() function.

Syntax

```
SQLRETURN SQLGetConnectOption (
    SQLHDBC          hdbc,
    SQLUSMALLINT    fOption,
    SQLPOINTER      pvParam);
```

Function arguments

Table 75. SQLGetConnectOption arguments

Data type	Argument	Use	Description
HDBC	hdbc	input	Connection handle.
SQLUSMALLINT	fOption	input	Option to set. See Table 136 on page 337 for the complete list of connection options and their descriptions.
SQLPOINTER	pvParam	input/output	Value associated with <i>fOption</i> . Depending on the value of <i>fOption</i> , this can be a 32-bit integer value, or a pointer to a null terminated character string. The maximum length of any character string returned is SQL_MAX_OPTION_STRING_LENGTH bytes (excluding the null-terminator).

Usage

If SQLGetConnectOption() is called, and the specified *fOption* has not been set using SQLSetConnectOption and does not have a default, then SQLGetConnectOption() returns SQL_NO_DATA_FOUND.

Although SQLSetConnectOption() can be used to set statement options, SQLGetConnectOption() cannot be used to retrieve statement options, use SQLGetStmtOption() instead.

For a list of valid connect options, see Table 136 on page 337, in the function description for SQLSetConnectAttr().

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 76. *SQLGetConnectOption* SQLSTATEs

SQLSTATE	Description	Explanation
40003 08S01	Communication link failure.	The function is called after the communication link source to which DB2 ODBC is connected, failed during the processing of a previous request.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	The <i>pvParam</i> argument is NULL.
S1092	Option type out of range.	An invalid <i>fOption</i> value is specified.
S1C00	Driver not capable.	The <i>fOption</i> is recognized, but is not supported.

Restrictions

None.

Example

```

/* ... */
rc = SQLGetConnectOption(hdbc, SQL_AUTOCOMMIT, &autocommit);
printf("Autocommit is: ");
if (autocommit == SQL_AUTOCOMMIT_ON)
    printf("ON\n");
else
    printf("OFF\n");
/* ... */

```

References

- “SQLSetConnectOption - Set connection option” on page 345
- “SQLSetStmtOption - Set statement option” on page 367
- “SQLGetStmtOption - Returns current setting of a statement option” on page 273

SQLGetCursorName - Get cursor name

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLGetCursorName() returns the cursor name associated with the input statement handle. If a cursor name is explicitly set by calling SQLSetCursorName(), this name is returned; otherwise, an implicitly generated name is returned.

Syntax

```
SQLRETURN SQLGetCursorName (SQLHSTMT      hstmt,
                             SQLCHAR       FAR *szCursor,
                             SQLSMALLINT   cbCursorMax,
                             SQLSMALLINT   FAR *pcbCursor);
```

Function arguments

Table 77. SQLGetCursorname arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle
SQLCHAR *	<i>szCursor</i>	output	Cursor name
SQLSMALLINT	<i>cbCursorMax</i>	input	Length of buffer <i>szCursor</i>
SQLSMALLINT *	<i>pcbCursor</i>	output	Number of bytes available to return for <i>szCursor</i>

Usage

SQLGetCursorName() returns the cursor name set explicitly with SQLSetCursorName(), or if no name is set, it returns the cursor name internally generated by DB2 ODBC.

If a name is set explicitly using SQLSetCursorName(), this name is returned until the statement is dropped, or until another explicit name is set.

Internally generated cursor names always begin with SQLCUR or SQL_CUR. For query result sets, DB2 ODBC also reserves SQLCURQRS as a cursor name prefix. Cursor names are always 18 characters or less, and are always unique within a connection.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 78. SQLGetCursorName SQLSTATES

SQLSTATE	Description	Explanation
01004	Data truncated.	The cursor name returned in <i>szCursor</i> is longer than the value in <i>cbCursorMax</i> , and is truncated to <i>cbCursorMax</i> - 1 bytes. The argument <i>pcbCursor</i> contains the length of the full cursor name available for return. The function returns SQL_SUCCESS_WITH_INFO.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY015	No cursor name available.	There is no open cursor on the statement handle specified by <i>hstmt</i> and no cursor name is set with SQLSetCursorName().
HY090	Invalid string or buffer length.	The value specified for the argument <i>cbCursorMax</i> is less than 0.
HY092	Option type out of range.	The value specified for the argument <i>hstmt</i> is not valid.

Restrictions

ODBC generated cursor names begin with SQL_CUR. X/Open CLI generated cursor names begin with either SQLCUR or SQL_CUR. DB2 ODBC also generates a cursor name that begins with SQLCUR or SQL_CUR.

Example

```

/*****
/* 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";

```

SQLGetCursorName

```
SWWORD      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 %s.\n");

rc = SQLBindCol (hStmt,          // bind employee name
                1,
                SQL_C_CHAR,
                empToyee,
                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;
}

```

SQLGetCursorName

```

/*****
/* Answer set is available -- Fetch rows and update salary */
*****/

while (((rc = SQLFetch (hStmt)) == SQL_SUCCESS) &&
      (rc2 == SQL_SUCCESS))
{
    int new_salary = salary*1.1;

    (void) printf ("**** Employee Name %s with salary %d. New salary = %d.\n",
                  employee,
                  salary,
                  new_salary);

    sprintf (update,
            "UPDATE EMPLOYEE SET SALARY = %d WHERE CURRENT OF %s",
            new_salary,
            cursor_name);

    (void) printf ("***** Update statement is : %s\n", update);

    rc2 = SQLExecDirect (hStmt2,
                        (SQLCHAR *) update,
                        SQL_NTS);
}

if (rc2 != SQL_SUCCESS)
{
    (void) printf ("**** EMPLOYEE UPDATE FAILED.\n");
    goto dberror;
}

/*****
/* Reexecute query to validate that salary was updated */
*****/

rc = SQLCloseCursor (hStmt);

rc = SQLExecDirect (hStmt,
                  (SQLCHAR *) stmt,
                  strlen(stmt));

if (rc != SQL_SUCCESS)
{
    (void) printf ("**** EMPLOYEE QUERY FAILED.\n");
    goto dberror;
}

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
    (void) printf ("**** Employee Name %s has salary %d.\n",
                  employee,
                  salary);
}

```

```

/*****
/* 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;
}

```

References

- “SQLExecute - Execute a statement” on page 166
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLPrepare - Prepare a statement” on page 300
- “SQLSetCursorName - Set cursor name” on page 347

SQLGetData - Get data from a column

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

SQLGetData() retrieves data for a single column in the current row of the result set. This is an alternative to SQLBindCol(), which is used to transfer data directly into application variables or LOB locators on each SQLFetch() or SQLExtendedFetch() call. SQLGetData() can also be used to retrieve large data values in pieces.

SQLFetch() must be called before SQLGetData().

After calling SQLGetData() for each column, SQLFetch() or SQLExtendedFetch() is called to retrieve the next row.

Syntax

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

Function arguments

Table 79. SQLGetData arguments

Data Type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle.
SQLUSMALLINT	<i>icol</i>	input	Column number for which the data retrieval is requested.
SQLSMALLINT	<i>fCType</i>	input	The C data type of the column identifier by <i>icol</i> . The following types are supported: <ul style="list-style-type: none"> • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_WCHAR Specifying SQL_C_DEFAULT results in the data being converted to its default C data type, see Table 4 on page 31 for more information.
SQLPOINTER	<i>rgbValue</i>	output	Pointer to buffer where the retrieved column data is to be stored.
SQLINTEGER	<i>cbValueMax</i>	input	Maximum size of the buffer pointed to by <i>rgbValue</i> .

Table 79. SQLGetData arguments (continued)

Data Type	Argument	Use	Description
SQLINTEGER *	<i>pcbValue</i>	output	<p>Pointer to value which indicates the number of bytes DB2 ODBC has available to return in the <i>rgbValue</i> buffer. If the data is being retrieved in pieces, this contains the number of bytes still remaining.</p> <p>The value is SQL_NULL_DATA if the data value of the column is null. If this pointer is NULL and SQLFetch() has obtained a column containing null data, then this function fails because it has no means of reporting this.</p> <p>If SQLFetch() has fetched a column containing binary data, then the pointer to <i>pcbValue</i> must not be NULL or this function fails because it has no other means of informing the application about the length of the data retrieved in the <i>rgbValue</i> buffer.</p>

Note: DB2 ODBC provides some performance enhancement if *rgbValue* is placed consecutively in memory after *pcbValue*.

Usage

SQLGetData() can be used with SQLBindCol() for the same result set, as long as SQLFetch() and not SQLExtendedFetch() is used. The general steps are:

1. SQLFetch() - advances cursor to first row, retrieves first row, transfers data for bound columns.
2. SQLGetData() - transfers data for the specified column.
3. Repeat step 2 for each column needed.
4. SQLFetch() - advances cursor to next row, retrieves next row, transfers data for bound columns.
5. Repeat steps 2, 3 and 4 for each row in the result set, or until the result set is no longer needed.

SQLGetData() can also be used to retrieve long columns if the C data type (*fCType*) 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.

To retrieve UCS-2 data, set *fCType* to SQL_C_WCHAR.

Upon each SQLGetData() call, if the data available for return is greater than or equal to *cbValueMax*, truncation occurs. Truncation is indicated by a function return code of SQL_SUCCESS_WITH_INFO coupled with a SQLSTATE denoting data truncation. The application can call SQLGetData() again, with the same *icol* value, to get subsequent data from the same unbound column starting at the point of truncation. To obtain the entire column, the application repeats such calls until the function returns SQL_SUCCESS. The next call to SQLGetData() returns SQL_NO_DATA_FOUND.

Although SQLGetData() can be used for the sequential retrieval of LOB column data, use the DB2 ODBC LOB functions if only a portion of the LOB data or a few sections of the LOB column data are needed:

1. Bind the column to a LOB locator.
2. Fetch the row.

SQLGetData

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.

Truncation is also affected by the `SQL_MAX_LENGTH` statement option. The application can specify that truncation is not to be reported by calling `SQLSetStmtAttr()` with `SQL_MAX_LENGTH` and a value for the maximum length to return for any one column, and by allocating a *rgbValue* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, `SQL_SUCCESS` is returned and the maximum length, not the actual length is returned in *pcbValue*.

To discard the column data part way through the retrieval, the application can call `SQLGetData()` with *icol* set to the next column position of interest. To discard data that has not been retrieved for the entire row, the application should call `SQLFetch()` to advance the cursor to the next row; or, if it is not interested in any more data from the result set, call `SQLFreeStmt()` or `SQLCloseCursor()` to close the cursor.

The *fCType* input argument determines the type of data conversion (if any) needed before the column data is placed into the storage area pointed to by *rgbValue*.

For SQL graphic column data:

- The length of the *rgbValue* buffer (*cbValueMax*) should be a multiple of 2. The application can determine the SQL data type of the column by first calling `SQLDescribeCol()` or `SQLColAttribute()`.
- The pointer to *pcbValue* must not be NULL since DB2 ODBC stores the number of octets stored in *rgbValue*.
- If the data is retrieved in piecewise fashion, DB2 ODBC attempts to fill *rgbValue* to the nearest multiple of two octets that is still less than or equal to *cbValueMax*. This means if *cbValueMax* is not a multiple of two, the last byte in that buffer is untouched; DB2 ODBC does not split a double-byte character.

The contents returned in *rgbValue* are always null-terminated unless the column data to be retrieved is binary, or if the SQL data type of the column is graphic (DBCS) and the C buffer type is `SQL_C_CHAR`. If the application is retrieving the data in multiple chunks, it should make the proper adjustments (for example, strip off the null-terminator before concatenating the pieces back together assuming the null termination environment attribute is in effect).

Truncation of numeric data types is reported as a warning if the truncation involves digits to the right of the decimal point. If truncation occurs to the left of the decimal point, an error is returned (see the 'Diagnostics' section).

Applications that use `SQLExtendedFetch()` to retrieve data should call `SQLGetData()` only when the rowset size is 1 (equivalent to issuing `SQLFetch()`). `SQLGetData()` can only retrieve column data for a row where the cursor is currently positioned.

Return codes

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`
- `SQL_NO_DATA_FOUND`

SQL_NO_DATA_FOUND is returned when the preceding SQLGetData() call has retrieved all of the data for this column.

SQL_SUCCESS is returned if a zero-length string is retrieved by SQLGetData(). If this is the case, *pcbValue* contains 0, and *rgbValue* contains a null terminator.

If the preceding call to SQLFetch() failed, SQLGetData() should not be called since the result is undefined.

Diagnostics

Table 80. SQLGetData SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	Data returned for the specified column (<i>icol</i>) is truncated. String or numeric values are right truncated. SQL_SUCCESS_WITH_INFO is returned.
07006	Invalid conversion.	The data value cannot be converted to the C data type specified by the argument <i>fcType</i> . The function has been called before for the same <i>icol</i> value but with a different <i>fcType</i> value.
07009	Invalid column number.	The specified column is less than 0 or greater than the number of result columns. The specified column is 0, but DB2 ODBC does not support ODBC bookmarks (<i>icol</i> = 0). SQLExtendedFetch() is called for this result set.
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument <i>pcbValue</i> is a null pointer and the value of the column is null. There is no means to report SQL_NULL_DATA.
22008	Invalid datetime format or datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22018	Error in assignment.	A returned value is incompatible with the data type denoted by the argument <i>fcType</i> .
24000	Invalid cursor state.	The previous SQLFetch() resulted in SQL_ERROR or SQL_NO_DATA found; as a result, the cursor is not positioned on a row.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	<i>fcType</i> is not a valid data type or SQL_C_DEFAULT.
HY009	Invalid use of a null pointer.	The argument <i>rgbValue</i> is a null pointer. The argument <i>pcbValue</i> is a null pointer; the column SQL data type is graphic (DBCS); and <i>fcType</i> is set to SQL_C_CHAR.

SQLGetData

Table 80. SQLGetData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The specified <i>hstmt</i> is not in a cursor positioned state. The function is called without first calling <code>SQLFetch()</code> . The function is called while in a data-at-execute (<code>SQLParamData()</code> , <code>SQLPutData()</code>) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY019	Numeric value out of range.	Returning the numeric value (as numeric or string) for the column causes the whole part of the number to be truncated.
HY090	Invalid string or buffer length.	The value of the argument <i>cbValueMax</i> is less than 0 and the argument <i>fCType</i> is <code>SQL_C_CHAR</code> , <code>SQL_C_BINARY</code> , <code>SQL_C_DBCHAR</code> or (<code>SQL_C_DEFAULT</code> and the default type is one of <code>SQL_C_CHAR</code> , <code>SQL_C_BINARY</code> , or <code>SQL_C_DBCHAR</code>).
HYC00	Driver not capable.	The SQL data type for the specified data type is recognized but not supported by DB2 ODBC. The requested conversion from the SQL data type to the application data <i>fCType</i> cannot be performed by DB2 ODBC or the data source. <code>SQLExtendedFetch()</code> is called for the specified <i>hstmt</i> .

Restrictions

ODBC has defined column 0 for bookmarks. DB2 ODBC does not support bookmarks.

Example

See “Example” on page 180 for a comparison between using bound columns and using `SQLGetData()`.

```

/*****
/*      Populate BIOGRAPHY table from flat file text. Insert      */
/*      VITAE in 80-byte pieces via SQLPutData. Also retrieve    */
/*      NAME, UNIT and VITAE for all members. VITAE is retrieved*/
/*      via SQLGetData.                                          */
*****/

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

#define TEXT_SIZE 80

int insert_bio (SQLHSTMT hStmt,      // insert_bio prototype
               char      *bio,
               int       bcount);

int main( )
{
    SQLHENV      hEnv   = SQL_NULL_HENV;
    SQLHDBC      hDbc   = SQL_NULL_HDBC;
    SQLHSTMT     hStmt  = SQL_NULL_HSTMT, hStmt2 = SQL_NULL_HSTMT;
    SQLRETURN    rc     = SQL_SUCCESS;
    FILE         *fp;
    SQLINTEGER   RETCODE = 0;
    char         pTable [200];
    char         *pDSN = "STLEC1";
    UWORD        pirow;
    SDWORD       cbValue;

```

SQLGetData

```
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;
}

```

SQLGetData

```
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;
}

```

SQLGetData

```
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 %s. Unit is %s.\n\n", name, unit);
    (void) printf ("**** Vitae follows:\n\n");

    for (i = 0; (i < 3200 && RETCODE != SQL_NO_DATA_FOUND); i += TEXT_SIZE)
    {
        RETCODE = SQLGetData (hStmt2,
                            3,
                            SQL_C_CHAR,
                            Narrative,
                            sizeof(Narrative) + 1,
                            &vitae_cbValue);

        if (RETCODE != SQL_NO_DATA_FOUND)
            (void) printf ("%s\n", Narrative);
    }
}

/*****
/* Deallocate Statement Handles */
*****/

rc =SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt);

rc =SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt2);
```

```

/*****
/* DISCONNECT from data source */
*****/

    RETCODE = SQLDisconnect(hDbc);

    if (RETCODE != SQL_SUCCESS)
        goto dberror;

/*****
/* Deallocate Connection Handle */
*****/

    RETCODE =SQLFreeHandle(SQL_HANDLE_DBC, hDbc);

    if (RETCODE != SQL_SUCCESS)
        goto dberror;

/*****
/* Free Environment Handle */
*****/

    RETCODE =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;

```

SQLGetData

```

/*****
/* 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);
}

```

References

- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169
- “SQLFetch - Fetch next row” on page 176

SQLGetDiagRec - Get multiple field settings of diagnostic record

Purpose

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

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

Syntax

```
SQLRETURN SQLGetDiagRec (SQLSMALLINT SQLHANDLE
                        SQLSMALLINT SQLCHAR
                        SQLINTEGER SQLCHAR
                        SQLSMALLINT SQLSMALLINT
                        HandleType,
                        Handle,
                        RecNumber,
                        *SQLState,
                        *NativeErrorPtr,
                        *MessageText,
                        BufferLength,
                        *TextLengthPtr);
```

Function arguments

Table 81. SQLGetDiagRec arguments

Data type	Argument	Use	Description
SQLSMALLINT	<i>HandleType</i>	input	A handle type identifier that describes the type of handle for which diagnostics are desired. Must be one of the following: <ul style="list-style-type: none"> SQL_HANDLE_ENV SQL_HANDLE_DBC SQL_HANDLE_STMT
SQLHANDLE	<i>Handle</i>	input	A handle for the diagnostic data structure, of the type indicated by <i>HandleType</i> .
SQLSMALLINT	<i>RecNumber</i>	input	Indicates the status record from which the application seeks information. Status records are numbered from 1.
SQLCHAR *	<i>SQLState</i>	output	Pointer to a buffer in which to return a five-character SQLSTATE code pertaining to the diagnostic record <i>RecNumber</i> . The first two characters indicate the class; the next three indicate the subclass.
SQLINTEGER *	<i>NativeErrorPtr</i>	output	Pointer to a buffer in which to return the native error code, specific to the data source.
SQLCHAR *	<i>MessageText</i>	output	Pointer to a buffer in which to return the error message text. The fields returned by SQLGetDiagRec() are contained in a text string.
SQLSMALLINT	<i>BufferLength</i>	input	Length (in bytes) of the <i>*MessageText</i> buffer.
SQLSMALLINT *	<i>TextLengthPtr</i>	output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes required for the null termination character) available to return in <i>*MessageText</i> . If the number of bytes available to return is greater than <i>BufferLength</i> , the error message text in <i>*MessageText</i> is truncated to <i>BufferLength</i> minus the length of a null termination character.

SQLGetDiagRec

Usage

An application typically calls `SQLGetDiagRec()` when a previous call to a DB2 ODBC function has returned anything other than `SQL_SUCCESS`. However, since 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 ODBC 2.0 functions `SQLGetDiagRec()`), any diagnostic information from the previous calls on the same handle is lost.

An application can scan all diagnostic records by looping, incrementing `RecNumber`, as long as `SQLGetDiagRec()` returns `SQL_SUCCESS`.

Calls to `SQLGetDiagRec()` are non-destructive to the 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 ODBC 2.0 functions `SQLGetDiagRec()`), has been called in the interim.

Return codes

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_INVALID_HANDLE`
- `SQL_ERROR`

Diagnostics

`SQLGetDiagRec()` does not post error values. It uses the following return values to report the outcome of its own execution:

- `SQL_SUCCESS`: The function successfully returned diagnostic information.
- `SQL_SUCCESS_WITH_INFO`: The **MessageText* buffer was too small to hold the requested diagnostic message. No diagnostic records were generated. To determine that a truncation occurred, the application must compare *BufferLength* to the actual number of bytes available, which is written to **StringLengthPtr*.
- `SQL_INVALID_HANDLE`: The handle indicated by *HandleType* and *Handle* was not a valid handle.
- `SQL_ERROR`: One of the following occurred:
 - *RecNumber* was negative or 0.
 - *BufferLength* was less than zero.
- `SQL_NO_DATA`: *RecNumber* was greater than the number of diagnostic records that existed for the handle specified in *Handle*. The function also returns `SQL_NO_DATA` for any positive *RecNumber* if there are no diagnostic records for *Handle*.

Restrictions

None.

Example

Refer to sample program `DSN8O3VP` in `DSN710.SDSNSAMP`.

References

- “SQLGetInfo - Get general information” on page 234
- “SQLFreeHandle - Free handle resources” on page 193
- “SQLFreeStmt - Free (or reset) a statement handle” on page 196

SQLGetEnvAttr - Returns current setting of an environment attribute

Purpose

Specification:		X/OPEN CLI	ISO CLI
----------------	--	------------	---------

SQLGetEnvAttr() returns the current setting for the specified environment attribute. These options are set using the SQLSetEnvAttr() function.

Syntax

```
SQLRETURN SQLGetEnvAttr (SQLHENV           EnvironmentHandle,
                        SQLINTEGER        Attribute,
                        SQLPOINTER       ValuePtr,
                        SQLINTEGER        BufferLength,
                        SQLINTEGER        *StringLengthPtr);
```

Function arguments

Table 82. SQLGetEnvAttr arguments

Data type	Argument	Use	Description
SQLHENV	<i>EnvironmentHandle</i>	input	Environment handle.
SQLINTEGER	<i>Attribute</i>	input	Attribute to retrieve. See Table 145 on page 351 for the list of environment attributes and their descriptions.
SQLPOINTER	<i>ValuePtr</i>	output	The current value associated with <i>Attribute</i> . The type of the value returned depends on <i>Attribute</i> .
SQLINTEGER	<i>BufferLength</i>	input	Maximum size of buffer pointed to by <i>ValuePtr</i> . <ul style="list-style-type: none"> If <i>ValuePtr</i> points to a character string, this argument should be the length of <i>*ValuePtr</i> or SQL_NTS. If SQL_NTS, the driver assumes that the length of <i>*ValuePtr</i> is SQL_MAX_OPTIONS_STRING_LENGTH bytes (excluding the null-terminator). If <i>ValuePtr</i> points to an integer, <i>BufferLength</i> is ignored.
SQLINTEGER *	<i>StringLengthPtr</i>	output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes returned for the null-termination character) available to return in <i>ValuePtr</i> . If <i>ValuePtr</i> 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 <i>BufferLength</i> , the data in <i>ValuePtr</i> is truncated to <i>BufferLength</i> minus the length of a null-termination character and is null-terminated by DB2 ODBC. <p>If <i>Attribute</i> does not denote a string, then DB2 ODBC ignores <i>BufferLength</i> and does not set <i>StringLengthPtr</i>.</p>

Usage

SQLGetEnvAttr() can be called at any time between the allocation and freeing of the environment handle. It obtains the current value of the environment attribute.

For a list of valid environment attributes, see Table 145 on page 351.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 83. SQLGetEnvAttr SQLSTATEs

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY092	Option type out of range.	An invalid <i>Attribute</i> value was specified.

Restrictions

None.

Example

```
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");
```

References

- “SQLSetEnvAttr - Set environment attribute” on page 350
- “SQLAllocHandle - Allocate handle” on page 79

SQLGetFunctions - Get functions

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLGetFunctions() to query whether a specific function is supported. This allows applications to adapt to varying levels of support when connecting to different database servers.

A connection to a database server must exist before calling this function.

Syntax

```
SQLRETURN SQLGetFunctions (SQLHDBC          hdbc,
                           SQLUSMALLINT     fFunction,
                           SQLUSMALLINT FAR *pfExists);
```

Function arguments

Table 84. SQLGetFunctions arguments

Data type	Argument	Use	Description
SQLHDBC	<i>hdbc</i>	input	Database connection handle.
SQLUSMALLINT	<i>fFunction</i>	input	The function being queried. Valid <i>fFunction</i> values are shown in Figure 7 on page 229
SQLUSMALLINT *	<i>pfExists</i>	output	Pointer to location where this function returns SQL_TRUE or SQL_FALSE depending on whether the function being queried is supported.

Usage

Figure 7 on page 229 shows the valid values for the *fFunction* argument and whether the corresponding function is supported.

If *fFunction* is set to SQL_API_ALL_FUNCTIONS, then *pfExists* must point to an SQLSMALLINT array of 100 elements. The array is indexed by the *fFunction* values used to identify many of the functions. Some elements of the array are unused and reserved. Since some *fFunction* values are greater than 100, the array method can not be used to obtain a list of functions. The SQLGetFunction() call must be explicitly issued for all *fFunction* values equal to or above 100. The complete set of *fFunction* values is defined in sqlcli1.h.

SQL_API_SQLALLOCCONNECT	= TRUE	SQL_API_SQLALLOCENV	= TRUE
SQL_API_SQLALLOCHANDLE	= TRUE	SQL_API_SQLALLOCSTMT	= TRUE
SQL_API_SQLBINDCOL	= TRUE	SQL_API_SQLBINDFILETOCOL	= FALSE
SQL_API_SQLBINDFILETOPARAM	= FALSE	SQL_API_SQLBINDPARAMETER	= TRUE
SQL_API_SQLBROWSECONNECT	= FALSE	SQL_API_SQLCANCEL	= TRUE
SQL_API_SQLCLOSECURSOR	= TRUE	SQL_API_SQLCOLATTRIBUTE	= TRUE
SQL_API_SQLCOLATTRIBUTES	= TRUE	SQL_API_SQLCOLUMNPRIVILEGES	= TRUE
SQL_API_SQLCOLUMNS	= TRUE	SQL_API_SQLCONNECT	= TRUE
SQL_API_SQLDATASOURCES	= TRUE	SQL_API_SQLDESCRIBECOL	= TRUE
SQL_API_SQLDESCRIBEPARAM	= TRUE	SQL_API_SQLDISCONNECT	= TRUE
SQL_API_SQLDRIVERCONNECT	= TRUE	SQL_API_SQLENDTRAN	= TRUE
SQL_API_SQLError	= TRUE	SQL_API_SQLEXECDIRECT	= TRUE
SQL_API_SQLEXECUTE	= TRUE	SQL_API_SQLEXTENDEDFETCH	= TRUE
SQL_API_SQLFETCH	= TRUE	SQL_API_SQLFOREIGNKEYS	= TRUE
SQL_API_SQLFREECONNECT	= TRUE	SQL_API_SQLFREEENV	= TRUE
SQL_API_SQLFREEHANDLE	= TRUE	SQL_API_SQLFREESTMT	= TRUE
SQL_API_SQLGETCONNECTATTR	= TRUE	SQL_API_SQLGETCONNECTOPTION	= TRUE
SQL_API_SQLGETCURSORNAME	= TRUE	SQL_API_SQLGETDATA	= TRUE
SQL_API_SQLGETDIAGREC	= TRUE	SQL_API_SQLGETENVATTR	= TRUE
SQL_API_SQLGETFUNCTIONS	= TRUE	SQL_API_SQLGETINFO	= TRUE
SQL_API_SQLGETLENGTH	= TRUE	SQL_API_SQLGETPOSITION	= TRUE
SQL_API_SQLSQLGETSQLCA	= TRUE	SQL_API_SQLGETSTMATTR	= TRUE
SQL_API_SQLGETSTMTOPTION	= TRUE	SQL_API_SQLGETSUBSTRING	= TRUE
SQL_API_SQLGETTYPEINFO	= TRUE	SQL_API_SQLMORERESULTS	= TRUE
SQL_API_SQLNATIVESQL	= TRUE	SQL_API_SQLNUMPARAMS	= TRUE
SQL_API_SQLNUMRESULTCOLS	= TRUE	SQL_API_SQLPARAMDATA	= TRUE
SQL_API_SQLPARAMOPTIONS	= TRUE	SQL_API_SQLPREPARE	= TRUE
SQL_API_SQLPRIMARYKEYS	= TRUE	SQL_API_SQLPROCEDURECOLUMNS	= TRUE
SQL_API_SQLPROCEDURES	= TRUE	SQL_API_SQLPUTDATA	= TRUE
SQL_API_SQLROWCOUNT	= TRUE	SQL_API_SQLSETCOLATTRIBUTES	= TRUE
SQL_API_SQLSETCONNECTATTR	= TRUE	SQL_API_SQLSETCONNECTION	= TRUE
SQL_API_SQLSETCONNECTOPTION	= TRUE	SQL_API_SQLSETCURSORNAME	= TRUE
SQL_API_SQLSETENVATTR	= TRUE	SQL_API_SQLSETPARAM	= TRUE
SQL_API_SQLSETPOS	= FALSE	SQL_API_SQLSETSCROLLOPTIONS	= FALSE
SQL_API_SQLSETSTMATTR	= TRUE	SQL_API_SQLSETSTMTOPTION	= TRUE
SQL_API_SQLSPECIALCOLUMNS	= TRUE	SQL_API_SQLSTATISTICS	= TRUE
SQL_API_SQLTABLEPRIVILEGES	= TRUE	SQL_API_SQLTABLES	= TRUE
SQL_API_TRANSACT	= TRUE		

Figure 7. Supported functions list

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 85. SQLGetFunctions SQLSTATEs

SQLSTATE	Description	Explanation
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	The argument pfExists was a null pointer.

SQLGetFunctions

Table 85. *SQLGetFunctions SQLSTATEs (continued)*

SQLSTATE	Description	Explanation
HY010	Function sequence error.	SQLGetFunctions() was called before a database connection was established.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

```

/*****
/* 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_SQLEXCEDIRECT   , "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");

```



```

/*****
/* 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);
}

```

References

None.

SQLGetInfo - Get general information

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLGetInfo() returns general information, (including supported data conversions) about the DBMS that the application is currently connected to.

Syntax

```
SQLRETURN SQLGetInfo (SQLHDBC          ConnectionHandle,
                     SQLUSMALLINT    InfoType,
                     SQLPOINTER       InfoValuePtr,
                     SQLSMALLINT      BufferLength,
                     SQLSMALLINT      *FAR StringLengthPtr);
```

Function arguments

Table 86. SQLGetInfo arguments

Data type	Argument	Use	Description
SQLHDBC	<i>ConnectionHandle</i>	input	Connection handle
SQLUSMALLINT	<i>InfoType</i>	input	The type of information requested. The argument must be one of the values in the first column of Table 87 on page 235.
SQLPOINTER	<i>InfoValuePtr</i>	output (also input)	Pointer to buffer where this function stores the desired information. Depending on the type of information being retrieved, 5 types of information can be returned: <ul style="list-style-type: none"> • 16-bit integer value • 32-bit integer value • 32-bit binary value • 32-bit mask • Null-terminated character string
SQLSMALLINT	<i>BufferLength</i>	input	Maximum length of the buffer pointed to by <i>InfoValuePtr</i> pointer.
SQLSMALLINT *	<i>StringLengthPtr</i>	output	Pointer to location where this function returns the total number of bytes available to return the desired information. In the case of string output, this size does not include the null terminating character. <p>If the value in the location pointed to by <i>StringLengthPtr</i> is greater than the size of the <i>InfoValuePtr</i> buffer as specified in <i>BufferLength</i>, the string output information is truncated to <i>BufferLength</i> - 1 bytes and the function returns with SQL_SUCCESS_WITH_INFO.</p>

Usage

Table 87 on page 235 lists the possible values of *InfoType* and a description of the information that SQLGetInfo() would return for that value. This table indicates which *InfoTypes* were renamed in ODBC 3.0.

Table 88 on page 255 lists the values of the *InfoType* arguments for SQLGetInfo() arguments that were renamed in ODBC 3.0.

Table 87. Information returned by SQLGetInfo

<i>InfoType</i>	<i>Format</i>	<i>Description and notes</i>
<p>Note: DB2 ODBC returns a value for each <i>InfoType</i> in this table. If the <i>InfoType</i> does not apply or is not supported, the result is dependent on the return type. If the return type is a:</p> <ul style="list-style-type: none"> • Character string containing 'Y' or 'N', "N" is returned. • Character string containing a value other than just 'Y' or 'N', an empty string is returned. • 16-bit integer, 0 (zero). • 32-bit integer, 0 (zero). • 32-bit mask, 0 (zero). 		
SQL_ACCESSIBLE_PROCEDURES	string	A character string of "Y" indicates that the user can execute all procedures returned by the function SQLProcedures(). "N" indicates that procedures can be returned that the user cannot execute.
SQL_ACCESSIBLE_TABLES	string	A character string of "Y" indicates that the user is guaranteed SELECT privilege to all tables returned by the function SQLTables(). "N" indicates that tables can be returned that the user cannot access.
SQL_ACTIVE_ENVIRONMENTS	16-bit integer	The maximum number of active environments that the DB2 ODBC driver can support. If there is no specified limit or the limit is unknown, this value is set to zero.
SQL_AGGREGATE_FUNCTIONS (32-bit mask)	32-bit mask	A bitmask enumerating support for aggregation functions: <ul style="list-style-type: none"> • SQL_AF_ALL • SQL_AF_AVG • SQL_AF_COUNT • SQL_AF_DISTINCT • SQL_AF_MAX • SQL_AF_MIN • SQL_AF_SUM
SQL_ALTER_DOMAIN	32-bit mask	DB2 ODBC returns 0 indicating that the ALTER DOMAIN statement is not supported. ODBC also defines the following values that DB2 ODBC does not return: <ul style="list-style-type: none"> • SQL_AD_ADD_CONSTRAINT_DEFERRABLE • SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE • SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED • SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE • SQL_AD_ADD_DOMAIN_CONSTRAINT • SQL_AD_ADD_DOMAIN_DEFAULT • SQL_AD_CONSTRAINT_NAME_DEFINITION • SQL_AD_DROP_DOMAIN_CONSTRAINT • SQL_AD_DROP_DOMAIN_DEFAULT
SQL_ALTER_TABLE	32-bit mask	Indicates which clauses in ALTER TABLE are supported by the DBMS. <ul style="list-style-type: none"> • SQL_AT_ADD_COLUMN • SQL_AT_DROP_COLUMN
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: <ul style="list-style-type: none"> • SQL_BRC_PROCEDURES • SQL_BRC_EXPLICIT

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

<i>InfoType</i>	Format	Description and notes
SQL_BATCH_SUPPORT	32-bit mask	Indicates which level of batches are supported: <ul style="list-style-type: none"> • SQL_BS_SELECT_EXPLICIT, supports explicit batches that can have result-set generating statements. • SQL_BS_ROW_COUNT_EXPLICIT, supports explicit batches that can have row-count generating statements. • SQL_BS_SELECT_PROC, supports explicit procedures that can have result-set generating statements. • SQL_BS_ROW_COUNT_PROC, supports explicit procedures that can have row-count generating statements.
SQL_BOOKMARK_PERSISTENCE	32-bit mask	Reserved option, zero is returned for the bit-mask.
SQL_CATALOG_LOCATION (In previous versions of DB2 ODBC this <i>InfoType</i> was SQL_QUALIFIER_LOCATION.)	16-bit integer	A 16-bit integer value indicated the position of the qualifier in a qualified table name. Zero indicates that qualified names are not supported.
SQL_CATALOG_NAME	string	A character string of 'Y' indicates that the server supports catalog names. 'N' indicates that catalog names are not supported.
SQL_CATALOG_NAME_SEPARATOR (In previous versions of DB2 ODBC this <i>InfoType</i> was SQL_QUALIFIER_NAME_SEPARATOR.)	string	The characters used as a separator between a catalog name and the qualified name element that follows it.
SQL_CATALOG_TERM (In previous versions of DB2 ODBC this <i>InfoType</i> was SQL_QUALIFIER_TERM.)	string	The database vendor's terminology for a qualifier. The name that the vendor uses for the high order part of a three part name. Since DB2 ODBC does not support three part names, a zero-length string is returned. For non-ODBC applications, the SQL_CATALOG_TERM symbolic name should be used instead of SQL_QUALIFIER_NAME.
SQL_CATALOG_USAGE (In previous versions of DB2 ODBC this <i>InfoType</i> was 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. <ul style="list-style-type: none"> • SQL_CB_NULL - indicates the result is a NULL value (this is the case for IBM RDBMs). • SQL_CB_NON_NULL - indicates the result is a concatenation of non-NULL column values.

Table 87. Information returned by SQLGetInfo (continued)

<i>InfoType</i>	<i>Format</i>	<i>Description and notes</i>
SQL_CONVERT_BIGINT SQL_CONVERT_BINARY SQL_CONVERT_BIT SQL_CONVERT_CHAR SQL_CONVERT_DATE SQL_CONVERT_DECIMAL SQL_CONVERT_DOUBLE SQL_CONVERT_FLOAT SQL_CONVERT_INTEGER SQL_CONVERT_INTERVAL_DAY_TIME SQL_CONVERT_INTERVAL_YEAR_MONTH SQL_CONVERT_LONGVARBINARY SQL_CONVERT_LONGVARCHAR SQL_CONVERT_NUMERIC SQL_CONVERT_REAL SQL_CONVERT_ROWID SQL_CONVERT_SMALLINT SQL_CONVERT_TIME SQL_CONVERT_TIMESTAMP SQL_CONVERT_TINYINT SQL_CONVERT_VARBINARY SQL_CONVERT_VARCHAR	32-bit mask	<p>Indicates the conversions supported by the data source with the CONVERT scalar function for data of the type named in the <i>infoType</i>. If the bitmask equals zero, the data source does not support any conversions for the data of the named type, including conversions to the same data type.</p> <p>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 <i>infoType</i> of SQL_CONVERT_INTEGER. The application then ANDs the returned bitmask with SQL_CVT_DECIMAL. If the resulting value is nonzero then the conversion is supported.</p> <p>The following bitmasks are used to determine which conversions are supported:</p> <ul style="list-style-type: none"> • SQL_CVT_BIGINT • SQL_CVT_BINARY • SQL_CVT_BIT • SQL_CVT_CHAR • SQL_CVT_DATE • SQL_CVT_DECIMAL • SQL_CVT_DOUBLE • SQL_CVT_FLOAT • SQL_CVT_INTEGER • SQL_CVT_LONGVARBINARY • SQL_CVT_LONGVARCHAR • SQL_CVT_NUMERIC • SQL_CVT_REAL • SQL_CVT_ROWID • SQL_CVT_SMALLINT • SQL_CVT_TIME • SQL_CVT_TIMESTAMP • SQL_CVT_TINYINT • SQL_CVT_VARBINARY • SQL_CVT_VARCHAR
SQL_CONVERT_FUNCTIONS	32-bit mask	<p>Indicates the scalar conversion functions supported by the driver and associated data source.</p> <ul style="list-style-type: none"> • SQL_FN_CVT_CONVERT - used to determine which conversion functions are supported. • SQL_FN_CVT_CAST - used to determine which cast functions are supported.
SQL_CORRELATION_NAME	16-bit integer	<p>Indicates the degree of correlation name support by the server:</p> <ul style="list-style-type: none"> • SQL_CN_ANY, supported and can be any valid user-defined name. • SQL_CN_NONE, correlation name not supported. • SQL_CN_DIFFERENT, correlation name supported but it must be different than the name of the table that it represents.

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_CLOSE_BEHAVIOR	32-bit integer	<p>Indicates whether or not locks are released when the cursor is closed. The possible values are:</p> <ul style="list-style-type: none"> • SQL_CC_NO_RELEASE: locks are not released when the cursor on this statement handle is closed. This is the default. • SQL_CC_RELEASE: locks are released when the cursor on this statement handle is closed. <p>Typically cursors are explicitly closed when the function SQLFreeStmt() is called with the SQL_CLOSE or SQL_DROP option. 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).</p>
SQL_CREATE_ASSERTION	32-bit mask	<p>Indicates which clauses in the CREATE ASSERTION statement are supported by the DBMS. DB2 ODBC always returns zero; the CREATE ASSERTION statement is not supported. ODBC also defines the following values that DB2 ODBC does not return:</p> <ul style="list-style-type: none"> • SQL_CA_CREATE_ASSERTION • SQL_CA_CONSTRAINT_INITIALLY_DEFERRED • SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE • SQL_CA_CONSTRAINT_DEFERRABLE • SQL_CA_CONSTRAINT_NON_DEFERRABLE
SQL_CREATE_CHARACTER_SET	32-bit mask	<p>Indicates which clauses in the CREATE CHARACTER SET statement are supported by the DBMS. DB2 ODBC always returns zero; the CREATE CHARACTER SET statement is not supported. ODBC also defines the following values that DB2 ODBC does not return:</p> <ul style="list-style-type: none"> • SQL_CCS_CREATE_CHARACTER_SET • SQL_CCS_COLLATE_CLAUSE • SQL_CCS_LIMITED_COLLATION
SQL_CREATE_COLLATION	32-bit mask	<p>Indicates which clauses in the CREATE COLLATION statement are supported by the DBMS. DB2 ODBC always returns zero; the CREATE COLLATION statement is not supported. ODBC also defines the following values that DB2 ODBC does not return:</p> <ul style="list-style-type: none"> • SQL_CCOL_CREATE_COLLATION
SQL_CREATE_DOMAIN	32-bit mask	<p>Indicates which clauses in the CREATE DOMAIN statement are supported by the DBMS. DB2 ODBC always returns zero; the CREATE DOMAIN statement is not supported. ODBC also defines the following values that DB2 ODBC does not return:</p> <ul style="list-style-type: none"> • SQL_CDO_CREATE_DOMAIN • SQL_CDO_CONSTRAINT_NAME_DEFINITION • SQL_CDO_DEFAULT • SQL_CDO_CONSTRAINT • SQL_CDO_COLLATION • SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED • SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE • SQL_CDO_CONSTRAINT_DEFERRABLE • SQL_CDO_CONSTRAINT_NON_DEFERRABLE
SQL_CREATE_SCHEMA	32-bit mask	<p>Indicates which clauses in the CREATE SCHEMA statement are supported by the DBMS:</p> <ul style="list-style-type: none"> • SQL_CS_CREATE_SCHEMA • SQL_CS_AUTHORIZATION • SQL_CS_DEFAULT_CHARACTER_SET

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_CREATE_TABLE	32-bit mask	<p>Indicates which clauses in the CREATE TABLE statement are supported by the DBMS. The following bitmasks are used to determine which clauses are supported:</p> <ul style="list-style-type: none"> • SQL_CT_CREATE_TABLE • SQL_CT_TABLE_CONSTRAINT • SQL_CT_CONSTRAINT_NAME_DEFINITION <p>The following bits specify the ability to create temporary tables:</p> <ul style="list-style-type: none"> • SQL_CT_COMMIT_PRESERVE, deleted rows are preserved on commit. • SQL_CT_COMMIT_DELETE, deleted rows are deleted on commit. • SQL_CT_GLOBAL_TEMPORARY, global temporary tables can be created. • SQL_CT_LOCAL_TEMPORARY, local temporary tables can be created. <p>The following bits specify the ability to create column constraints:</p> <ul style="list-style-type: none"> • SQL_CT_COLUMN_CONSTRAINT, specifying column constraints is supported. • SQL_CT_COLUMN_DEFAULT, specifying column defaults is supported. • SQL_CT_COLUMN_COLLATION, specifying column collation is supported. <p>The following bits specify the supported constraint attributes if specifying column or table constraints is supported:</p> <ul style="list-style-type: none"> • SQL_CT_CONSTRAINT_INITIALLY_DEFERRED • SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE • SQL_CT_CONSTRAINT_DEFERRABLE • SQL_CT_CONSTRAINT_NON_DEFERRABLE
SQL_CREATE_TRANSLATION	32-bit mask	<p>Indicates which clauses in the CREATE TRANSLATION statement are supported by the DBMS. DB2 ODBC always returns zero; the CREATE TRANSLATION statement is not supported. ODBC also defines the following value that DB2 ODBC does not return:</p> <ul style="list-style-type: none"> • SQL_CTR_CREATE_TRANSLATION
SQL_CURSOR_COMMIT_BEHAVIOR	16-bit integer	<p>Indicates how a COMMIT operation affects cursors. A value of:</p> <ul style="list-style-type: none"> • SQL_CB_DELETE, destroys cursors and drops access plans for dynamic SQL statements. • SQL_CB_CLOSE, destroys cursors, but retains access plans for dynamic SQL statements (including non-query statements) • SQL_CB_PRESERVE, retains cursors and access plans for dynamic statements (including non-query statements). Applications can continue to fetch data, or close the cursor and re-execute the query without re-preparing the statement. <p>After COMMIT, a FETCH must be issued to reposition the cursor before actions such as positioned updates or deletes can be taken.</p>

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_CURSOR_ROLLBACK_BEHAVIOR	16-bit integer	<p>Indicates how a ROLLBACK operation affects cursors. A value of:</p> <ul style="list-style-type: none"> SQL_CB_DELETE, destroys cursors and drops access plans for dynamic SQL statements. SQL_CB_CLOSE, destroys cursors, but retains access plans for dynamic SQL statements (including non-query statements) SQL_CB_PRESERVE, retains cursors and access plans for dynamic statements (including non-query statements). Applications can continue to fetch data, or close the cursor and re-execute the query without re-preparing the statement. <p>DB2 servers do not have the SQL_CB_PRESERVE property.</p>
SQL_CURSOR_SENSITIVITY	32-bit unsigned integer	<p>Indicates support for cursor sensitivity:</p> <ul style="list-style-type: none"> SQL_INSENSITIVE, all cursors on the statement handle show the result set without reflecting any changes made to it by any other cursor within the same transaction. SQL_UNSPECIFIED, it is unspecified whether cursors on the statement handle make visible the changes made to a result set by another cursor within the same transaction. Cursors on the statement handle may make visible none, some, or all such changes. SQL_SENSITIVE, cursors are sensitive to changes made by other cursors within the same transaction.
SQL_DATA_SOURCE_NAME	string	The name used as data source on the input to SQLConnect(), or the DSN keyword value in the SQLDriverConnect() connection string.
SQL_DATA_SOURCE_READ_ONLY	string	A character string of "Y" indicates that the database is set to READ ONLY mode; an "N" indicates that it is not set to READ ONLY mode.
SQL_DATABASE_NAME	string	The name of the current database in use. Note: Also returned by SELECT CURRENT SERVER on IBM DBMS's.
SQL_DBMS_NAME	string	The name of the DBMS product being accessed. For example: <ul style="list-style-type: none"> "DB2/6000" "DB2/2"
SQL_DBMS_VER	string	The Version of the DBMS product accessed. A string of the form 'mm.vv.rrrr' where mm is the major version, vv is the minor version and rrrr is the release. For example, "02.01.0000" translates to major version 2, minor version 1, release 0.
SQL_DDL_INDEX	32-bit unsigned integer	<p>Indicates support for the creation and dropping of indexes:</p> <ul style="list-style-type: none"> SQL_DI_CREATE_INDEX SQL_DI_DROP_INDEX

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_DEFAULT_TXN_ISOLATION	32-bit mask	<p>The default transaction isolation level supported.</p> <p>One of the following masks are returned:</p> <ul style="list-style-type: none"> • SQL_TXN_READ_UNCOMMITTED = Changes are immediately perceived by all transactions (dirty read, non-repeatable read, and phantoms are possible). This is equivalent to IBM's UR level. • SQL_TXN_READ_COMMITTED = Row read by transaction 1 can be altered and committed by transaction 2 (non-repeatable read and phantoms are possible) This is equivalent to IBM's CS level. • SQL_TXN_REPEATABLE_READ = A transaction can add or remove rows matching the search condition or a pending transaction (repeatable read, but phantoms are possible) This is equivalent to IBM's RS level. • SQL_TXN_SERIALIZABLE = Data affected by pending transaction is not available to other transactions (repeatable read, phantoms are not possible) This is equivalent to IBM's RR level. • SQL_TXN_VERSIONING = Not applicable to IBM DBMSs. • SQL_TXN_NOCOMMIT = Any changes are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed. This is a DB2 for AS/400 isolation level. <p>In IBM terminology,</p> <ul style="list-style-type: none"> • SQL_TXN_READ_UNCOMMITTED is uncommitted read; • SQL_TXN_READ_COMMITTED is cursor stability; • SQL_TXN_REPEATABLE_READ is read stability; • SQL_TXN_SERIALIZABLE is repeatable read.
SQL_DESCRIBE_PARAMETER	STRING	'Y' if parameters can be described; 'N' if not.
SQL_DRIVER_HDBC	32 bits	DB2 ODBC's current database handle.
SQL_DRIVER_HENV	32 bits	DB2 ODBC's environment handle.
SQL_DRIVER_HLIB	32 bits	Reserved.
SQL_DRIVER_HSTMT	32 bits	DB2 ODBC's current statement handle for the current connection.
SQL_DRIVER_NAME	string	The file name of the DB2 ODBC implementation. DB2 ODBC returns NULL.
SQL_DRIVER_ODBC_VER	string	The version number of ODBC that the Driver supports. DB2 ODBC returns "2.1".
SQL_DRIVER_VER	string	The version of the CLI driver. A string of the form 'mm.vv.rrrr' where mm is the major version, vv is the minor version and rrrr is the release. For example, "02.01.0000" translates to major version 2, minor version 1, release 0.
SQL_DROP_ASSERTION	32-bit mask	<p>Indicates which clause in the DROP ASSERTION statement is supported by the DBMS. DB2 ODBC always returns zero; the DROP ASSERTION statement is not supported. ODBC also defines the following value that DB2 ODBC does not return:</p> <ul style="list-style-type: none"> • SQL_DA_DROP_ASSERTION

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_DROP_CHARACTER_SET	32-bit mask	Indicates which clause in the DROP CHARACTER SET statement is supported by the DBMS. DB2 ODBC always returns zero; the DROP CHARACTER SET statement is not supported. ODBC also defines the following value that DB2 ODBC does not return.. <ul style="list-style-type: none"> • SQL_DCS_DROP_CHARACTER_SET
SQL_DROP_COLLATION	32-bit mask	Indicates which clause in the DROP COLLATION statement is supported by the DBMS. DB2 ODBC always returns zero; the DROP COLLATION statement is not supported. ODBC also defines the following value that DB2 ODBC does not return: <ul style="list-style-type: none"> • SQL_DC_DROP_COLLATION
SQL_DROP_DOMAIN	32-bit mask	Indicates which clauses in the DROP DOMAIN statement are supported by the DBMS. DB2 ODBC always returns zero; the DROP DOMAIN statement is not supported. ODBC also defines the following values that DB2 ODBC does not return: <ul style="list-style-type: none"> • SQL_DD_DROP_DOMAIN • SQL_DD_CASCADE • SQL_DD_RESTRICT
SQL_DROP_SCHEMA	32-bit mask	Indicates which clauses in the DROP SCHEMA statement are supported by the DBMS. <ul style="list-style-type: none"> • SQL_DS_DROP_SCHEMA • SQL_DS_CASCADE • SQL_DS_RESTRICT
SQL_DROP_TABLE	32-bit mask	Indicates which clauses in the DROP TABLE statement are supported by the DBMS: <ul style="list-style-type: none"> • SQL_DT_DROP_TABLE • SQL_DT_CASCADE • SQL_DT_RESTRICT
SQL_DROP_TRANSLATION	32-bit mask	Indicates which clauses in the DROP TRANSLATION statement are supported by the DBMS. DB2 ODBC always returns zero; the DROP TRANSLATION statement is not supported. ODBC also defines the following value that DB2 ODBC does not return: <ul style="list-style-type: none"> • SQL_DTR_DROP_TRANSLATION
SQL_DROP_VIEW	32-bit mask	Indicates which clauses in the DROP VIEW statement are supported by the DBMS. <ul style="list-style-type: none"> • SQL_DV_DROP_VIEW • SQL_DV_CASCADE • SQL_DV_RESTRICT

Table 87. 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). <ul style="list-style-type: none"> • SQL_CA1_NEXT • SQL_CA1_ABSOLUTE • SQL_CA1_RELATIVE • SQL_CA1_BOOKMARK • SQL_CA1_LOCK_EXCLUSIVE • SQL_CA1_LOCK_NO_CHANGE • SQL_CA1_LOCK_UNLOCK • SQL_CA1_POS_POSITION • SQL_CA1_POS_UPDATE • SQL_CA1_POS_DELETE • SQL_CA1_POS_REFRESH • SQL_CA1_POSITIONED_UPDATE • SQL_CA1_POSITIONED_DELETE • SQL_CA1_SELECT_FOR_UPDATE • SQL_CA1_BULK_ADD • SQL_CA1_BULK_UPDATE_BY_BOOKMARK • SQL_CA1_BULK_DELETE_BY_BOOKMARK • SQL_CA1_BULK_FETCH_BY_BOOKMARK
SQL_DYNAMIC_CURSOR_ATTRIBUTES2	32-bit mask	Indicates the attributes of a dynamic cursor that DB2 ODBC supports (subset 2 of 2). <ul style="list-style-type: none"> • SQL_CA2_READ_ONLY_CONCURRENCY • SQL_CA2_LOCK_CONCURRENCY • SQL_CA2_OPT_ROWVER_CONCURRENCY • SQL_CA2_OPT_VALUES_CONCURRENCY • SQL_CA2_SENSITIVITY_ADDITIONS • SQL_CA2_SENSITIVITY_DELETIONS • SQL_CA2_SENSITIVITY_UPDATES • SQL_CA2_MAX_ROWS_SELECT • SQL_CA2_MAX_ROWS_INSERT • SQL_CA2_MAX_ROWS_DELETE • SQL_CA2_MAX_ROWS_UPDATE • SQL_CA2_MAX_ROWS_CATALOG • SQL_CA2_MAX_ROWS_AFFECTS_ALL • SQL_CA2_CRC_EXACT • SQL_CA2_CRC_APPROXIMATE • SQL_CA2_SIMULATE_NON_UNIQUE • SQL_CA2_SIMULATE_TRY_UNIQUE • SQL_CA2_SIMULATE_UNIQUE
SQL_EXPRESSIONS_IN_ORDERBY	string	The character string "Y" indicates the database server supports the DIRECT specification of expressions in the ORDER BY list, "N" indicates that it does not.
SQL_FETCH_DIRECTION	32-bit mask	The supported fetch directions. The following bit-masks are used in conjunction with the flag to determine which options are supported. <ul style="list-style-type: none"> • SQL_FD_FETCH_NEXT • SQL_FD_FETCH_FIRST • SQL_FD_FETCH_LAST • SQL_FD_FETCH_PREV • SQL_FD_FETCH_ABSOLUTE • SQL_FD_FETCH_RELATIVE • SQL_FD_FETCH_RESUME
SQL_FILE_USAGE	16-bit integer	Reserved. Zero is returned.

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1	32-bit mask	<p>Indicates the attributes of a forward-only cursor that DB2 ODBC supports (subset 1 of 2).</p> <ul style="list-style-type: none"> • SQL_CA1_NEXT • SQL_CA1_POSITIONED_UPDATE • SQL_CA1_POSITIONED_DELETE • SQL_CA1_SELECT_FOR_UPDATE • SQL_CA1_LOCK_EXCLUSIVE • SQL_CA1_LOCK_NO_CHANGE • SQL_CA1_LOCK_UNLOCK • SQL_CA1_POS_POSITION • SQL_CA1_POS_UPDATE • SQL_CA1_POS_DELETE • SQL_CA1_POS_REFRESH • SQL_CA1_BULK_ADD • SQL_CA1_BULK_UPDATE_BY_BOOKMARK • SQL_CA1_BULK_DELETE_BY_BOOKMARK • SQL_CA1_BULK_FETCH_BY_BOOKMARK
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2	32-bit mask	<p>Indicates the attributes of a forward-only cursor that DB2 ODBC supports (subset 2 of 2).</p> <ul style="list-style-type: none"> • SQL_CA2_READ_ONLY_CONCURRENCY • SQL_CA2_LOCK_CONCURRENCY • SQL_CA2_MAX_ROWS_SELECT • SQL_CA2_MAX_ROWS_CATALOGQUE • SQL_CA2_OPT_ROWVER_CONCURRENCY • SQL_CA2_OPT_VALUES_CONCURRENCY • SQL_CA2_SENSITIVITY_ADDITIONS • SQL_CA2_SENSITIVITY_DELETIONS • SQL_CA2_SENSITIVITY_UPDATES • SQL_CA2_MAX_ROWS_INSERT • SQL_CA2_MAX_ROWS_DELETE • SQL_CA2_MAX_ROWS_UPDATE • SQL_CA2_MAX_ROWS_AFFECTS_ALL • SQL_CA2_CRC_EXACT • SQL_CA2_CRC_APPROXIMATE • SQL_CA2_SIMULATE_NON_UNIQUE • SQL_CA2_SIMULATE_TRY_UNIQUE • SQL_CA2_SIMULATE_UNIQUE
SQL_GETDATA_EXTENSIONS	32-bit mask	<p>Indicates whether extensions to the SQLGetData() function are supported. The following extensions are currently identified and supported by DB2 ODBC:</p> <ul style="list-style-type: none"> • SQL_GD_ANY_COLUMN, SQLGetData() can be called for unbound columns that precede the last bound column. • SQL_GD_ANY_ORDER, SQLGetData() can be called for columns in any order. <p>ODBC also defines SQL_GD_BLOCK and SQL_GD_BOUNDED; these bits are not returned by DB2 ODBC.</p>
SQL_GROUP_BY	16-bit integer	<p>Indicates the degree of support for the GROUP BY clause by the server:</p> <ul style="list-style-type: none"> • SQL_GB_NO_RELATION, there is no relationship between the columns in the GROUP BY and in the SELECT list • SQL_GB_NOT_SUPPORTED, GROUP BY not supported • SQL_GB_GROUP_BY_EQUALS_SELECT, GROUP BY must include all non-aggregated columns in the select list. • SQL_GB_GROUP_BY_CONTAINS_SELECT, the GROUP BY clause must contain all non-aggregated columns in the SELECT list.

Table 87. Information returned by SQLGetInfo (continued)

<i>InfoType</i>	Format	Description and notes
SQL_IDENTIFIER_CASE	16-bit integer	<p>Indicates case sensitivity of object names (such as table-name).</p> <p>A value of:</p> <ul style="list-style-type: none"> SQL_IC_UPPER = identifier names are stored in upper case in the system catalog. SQL_IC_LOWER = identifier names are stored in lower case in the system catalog. SQL_IC_SENSITIVE = identifier names are case sensitive, and are stored in mixed case in the system catalog. SQL_IC_MIXED = identifier names are not case sensitive, and are stored in mixed case in the system catalog. <p>Note: Identifier names in IBM DBMSs are not case sensitive.</p>
SQL_IDENTIFIER_QUOTE_CHAR	string	Indicates the character used to surround a delimited identifier.
SQL_INFO_SCHEMA_VIEWS	32-bit mask	<p>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:</p> <ul style="list-style-type: none"> SQL_ISV_ASSERTIONS SQL_ISV_CHARACTER_SETS SQL_ISV_CHECK_CONSTRAINTS SQL_ISV_COLLATIONS SQL_ISV_COLUMN_DOMAIN_USAGE SQL_ISV_COLUMN_PRIVILEGES SQL_ISV_COLUMNS SQL_ISV_CONSTRAINT_COLUMN_USAGE SQL_ISV_CONSTRAINT_TABLE_USAGE SQL_ISV_DOMAIN_CONSTRAINTS SQL_ISV_DOMAINS SQL_ISV_KEY_COLUMN_USAGE SQL_ISV_REFERENTIAL_CONSTRAINTS SQL_ISV_SCHEMATA SQL_ISV_SQL_LANGUAGES SQL_ISV_TABLE_CONSTRAINTS SQL_ISV_TABLE_PRIVILEGES SQL_ISV_TABLES SQL_ISV_TRANSLATIONS SQL_ISV_USAGE_PRIVILEGES SQL_ISV_VIEW_COLUMN_USAGE SQL_ISV_VIEW_TABLE_USAGE SQL_ISV_VIEWS
SQL_INSERT_STATEMENT	32-bit mask	<p>Indicates support for INSERT statements:</p> <ul style="list-style-type: none"> SQL_IS_INSERT_LITERALS SQL_IS_INSERT_SEARCHED SQL_IS_SELECT_INTO
SQL_INTEGRITY	string	<p>The "Y" character string indicates that the data source supports Integrity Enhanced Facility (IEF) in SQL89 and in X/Open XPG4 Embedded SQL; an "N" indicates it does not.</p>

(In previous versions of DB2 ODBC this *InfoType* was SQL_ODBC_SQL_OPT_IEF.)

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

<i>InfoType</i>	<i>Format</i>	<i>Description and notes</i>
SQL_KEYSET_CURSOR_ATTRIBUTES1	32-bit mask	Indicates the attributes of a keyset cursor that DB2 ODBC supports (subset 1 of 2). <ul style="list-style-type: none"> SQL_CA1_NEXT SQL_CA1_ABSOLUTE SQL_CA1_RELATIVE SQL_CA1_BOOKMARK SQL_CA1_LOCK_EXCLUSIVE SQL_CA1_LOCK_NO_CHANGE SQL_CA1_LOCK_UNLOCK SQL_CA1_POS_POSITION SQL_CA1_POS_UPDATE SQL_CA1_POS_DELETE SQL_CA1_POS_REFRESH SQL_CA1_POSITIONED_UPDATE SQL_CA1_POSITIONED_DELETE SQL_CA1_SELECT_FOR_UPDATE SQL_CA1_BULK_ADD SQL_CA1_BULK_UPDATE_BY_BOOKMARK SQL_CA1_BULK_DELETE_BY_BOOKMARK SQL_CA1_BULK_FETCH_BY_BOOKMARK
SQL_KEYSET_CURSOR_ATTRIBUTES2	32-bit mask	Indicates the attributes of a keyset cursor that DB2 ODBC supports (subset 2 of 2). <ul style="list-style-type: none"> SQL_CA2_READ_ONLY_CONCURRENCY SQL_CA2_LOCK_CONCURRENCY SQL_CA2_OPT_ROWVER_CONCURRENCY SQL_CA2_OPT_VALUES_CONCURRENCY SQL_CA2_SENSITIVITY_ADDITIONS SQL_CA2_SENSITIVITY_DELETIONS SQL_CA2_SENSITIVITY_UPDATES SQL_CA2_MAX_ROWS_SELECT SQL_CA2_MAX_ROWS_INSERT SQL_CA2_MAX_ROWS_DELETE SQL_CA2_MAX_ROWS_UPDATE SQL_CA2_MAX_ROWS_CATALOG SQL_CA2_MAX_ROWS_AFFECTS_ALL SQL_CA2_CRC_EXACT SQL_CA2_CRC_APPROXIMATE SQL_CA2_SIMULATE_NON_UNIQUE SQL_CA2_SIMULATE_TRY_UNIQUE SQL_CA2_SIMULATE_UNIQUE
SQL_KEYWORDS	string	This is a string of all the <i>keywords</i> at the DBMS that are not in the ODBC's list of reserved words.
SQL_LIKE_ESCAPE_CLAUSE	string	A character string that indicates if an escape character is supported for the metacharacters percent and underscore in a LIKE predicate.
SQL_LOCK_TYPES	32-bit mask	Reserved option, zero is returned for the bit-mask.
SQL_MAX_ASYNC_CONCURRENT_STATEMENTS	32-bit unsigned integer	The maximum number of active concurrent statements in asynchronous mode that DB2 ODBC can support on a given connection. This value is zero if there is no specific limit, or the limit is unknown.
SQL_MAX_BINARY_LITERAL_LEN	32-bit integer	A 32-bit integer value specifying the maximum length of a hexadecimal literal in a SQL statement.
SQL_MAX_CATALOG_NAME_LEN	16-bit integer	The maximum length of a catalog qualifier name; first part of a 3 part table name (in bytes).
(In previous versions of DB2 ODBC this <i>InfoType</i> was SQL_MAX_QUALIFIER_NAME_LEN.)		
SQL_MAX_CHAR_LITERAL_LEN	32-bit integer	The maximum length of a character literal in an SQL statement (in bytes).

Table 87. Information returned by SQLGetInfo (continued)

<i>InfoType</i>	Format	Description and notes
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 <i>InfoType</i> was SQL_ACTIVE_STATEMENTS.)	16-bit integer	The maximum number of active statements per connection. Zero is returned, indicating that the limit is dependent on database system and DB2 ODBC resources, and limits.
SQL_MAX_CURSOR_NAME_LEN	16-bit integer	The maximum length of a cursor name (in bytes).
SQL_MAX_DRIVER_CONNECTIONS (In previous versions of DB2 ODBC this <i>InfoType</i> was SQL_ACTIVE_CONNECTIONS.)	16-bit integer	The maximum number of active connections supported per application. Zero is returned, indicating that the limit is dependent on system resources. The MAXCONN keyword in the initialization file or the SQL_MAX_CONNECTIONS environment/connection option can be used to impose a limit on the number of connections. This limit is returned if it is set to any value other than zero.
SQL_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	Set to "Y" to indicate that the value returned by SQL_MAX_ROW_SIZE <i>InfoType</i> includes the length of product-specific <i>long string</i> data types. Otherwise, set to "N".
SQL_MAX_SCHEMA_NAME_LEN (In previous versions of DB2 ODBC this <i>InfoType</i> was 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 of an SQL statement string in bytes, including the number of white spaces in the statement.
SQL_MAX_TABLE_NAME_LEN	16-bit integer	The maximum length of a table name (in bytes).
SQL_MAX_TABLES_IN_SELECT	16-bit integer	Indicates the maximum number of table names allowed in a FROM clause in a <query specification>.
SQL_MAX_USER_NAME_LEN	16-bit integer	Indicates the maximum size allowed for a <user identifier> (in bytes).

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
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: <ul style="list-style-type: none"> • SQL_NNC_NON_NULL, columns can be defined as NOT NULL. • SQL_NNC_NULL, columns can not be defined as NOT NULL.
SQL_NULL_COLLATION	16-bit integer	Indicates where NULLs are sorted in a list: <ul style="list-style-type: none"> • SQL_NC_HIGH, null values sort high • SQL_NC_LOW, to indicate that null values sort low
SQL_NUMERIC_FUNCTIONS	32-bit mask	Indicates the ODBC scalar numeric functions supported. These functions are intended to be used with the ODBC vendor escape sequence described in "Using vendor escape clauses" on page 448. <p>The following bit-masks are used to determine which numeric functions are supported:</p> <ul style="list-style-type: none"> • SQL_FN_NUM_ABS • SQL_FN_NUM_ACOS • SQL_FN_NUM_ASIN • SQL_FN_NUM_ATAN • SQL_FN_NUM_ATAN2 • SQL_FN_NUM_CEILING • SQL_FN_NUM_COS • SQL_FN_NUM_COT • SQL_FN_NUM_DEGREES • SQL_FN_NUM_EXP • SQL_FN_NUM_FLOOR • SQL_FN_NUM_LOG • SQL_FN_NUM_LOG10 • SQL_FN_NUM_MOD • SQL_FN_NUM_PI • SQL_FN_NUM_POWER • SQL_FN_NUM_RADIANS • SQL_FN_NUM_RAND • SQL_FN_NUM_ROUND • SQL_FN_NUM_SIGN • SQL_FN_NUM_SIN • SQL_FN_NUM_SQRT • SQL_FN_NUM_TAN • SQL_FN_NUM_TRUNCATE
SQL_ODBC_API_CONFORMANCE	16-bit integer	The level of ODBC conformance. <ul style="list-style-type: none"> • SQL_OAC_NONE • SQL_OAC_LEVEL1 • SQL_OAC_LEVEL2
SQL_ODBC_SAG_CLI_CONFORMANCE	16-bit integer	The compliance to the functions of the SQL Access Group (SAG) CLI specification. <p>A value of:</p> <ul style="list-style-type: none"> • SQL_OSCC_NOT_COMPLIANT - the driver is not SAG-compliant. • SQL_OSCC_COMPLIANT - the driver is SAG-compliant.

Table 87. Information returned by SQLGetInfo (continued)

<i>InfoType</i>	Format	Description and notes
SQL_ODBC_SQL_CONFORMANCE	16-bit integer	<p>A value of:</p> <ul style="list-style-type: none"> SQL_OSC_MINIMUM - means minimum ODBC SQL grammar supported SQL_OSC_CORE - means core ODBC SQL Grammar supported SQL_OSC_EXTENDED - means extended ODBC SQL Grammar supported <p>For the definition of the above 3 types of ODBC SQL grammar, see <i>Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference</i>.</p>
SQL_ODBC_VER	string	<p>The version number of ODBC that the driver manager supports.</p> <p>DB2 ODBC returns the string "02.10".</p>
SQL_OJ_CAPABILITIES	32-bit mask	<p>A 32-bit bit-mask enumerating the types of outer join supported.</p> <p>The bitmasks are:</p> <ul style="list-style-type: none"> SQL_OJ_LEFT : Left outer join is supported. SQL_OJ_RIGHT : Right outer join is supported. SQL_OJ_FULL : Full outer join is supported. SQL_OJ_NESTED : Nested outer join is supported. SQL_OJ_NOT_ORDERED : The order of the tables underlying the columns in the outer join ON clause need not be in the same order as the tables in the JOIN clause. SQL_OJ_INNER : The inner table of an outer join can also be an inner join. SQL_OJ_ALL_COMPARISONS_OPS : Any predicate can be used in the outer join ON clause. If this bit is not set, the equality (=) operator is the only valid comparison operator in the ON clause.
SQL_ORDER_BY_COLUMNS_IN_SELECT	string	Set to "Y" if columns in the ORDER BY clauses must be in the select list; otherwise set to "N".
SQL_OUTER_JOINS	string	<p>The character string:</p> <ul style="list-style-type: none"> "Y" indicates that outer joins are supported, and DB2 ODBC supports the ODBC outer join request syntax. "N" indicates that it is not supported. <p>(See "Using vendor escape clauses" on page 448)</p>
SQL_OWNER_TERM	string	The database vendor's terminology for a schema (owner)

(In previous versions of DB2 ODBC this *InfoType* was SQL_SCHEMA_TERM.)

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_PARAM_ARRAY_ROW_COUNTS	32-bit unsigned integer	<p>Indicates the availability of row counts in a parameterized execution:</p> <ul style="list-style-type: none"> • SQL_PARC_BATCH: Individual row counts are available for each set of parameters. This is conceptually equivalent to the driver generating a batch of SQL statements, one for each parameter set in the array. Extended error information can be retrieved by using the SQL_PARAM_STATUS_PTR descriptor field. • SQL_PARC_NO_BATCH: Only one row count is available, which is the cumulative row count resulting from the execution of the statement for the entire array of parameters. This is conceptually equivalent to treating the statement along with the entire parameter array as one atomic unit. Errors are handled the same as if one statement were executed.
SQL_PARAM_ARRAY_SELECTS	32-bit unsigned integer	<p>Indicates the availability of result sets in a parameterized execution:</p> <ul style="list-style-type: none"> • SQL_PAS_BATCH: One result set is available per set of parameters. This is conceptually equivalent to the driver generating a batch of SQL statements, one for each parameter set in the array. • SQL_PAS_NO_BATCH: Only one result set is available, which represents the cumulative result set resulting from the execution of the statement for the entire array of parameters. This is conceptually equivalent to treating the statement along with the entire parameter array as one atomic unit. • SQL_PAS_NO_SELECT: A driver does not allow a result-set generating statement to be executed with an array of parameters.
SQL_POS_OPERATIONS	32-bit mask	Reserved option, zero is returned for the bit-mask.
SQL_POSITIONED_STATEMENTS	32-bit mask	<p>Indicates the degree of support for positioned UPDATE and positioned DELETE statements:</p> <ul style="list-style-type: none"> • SQL_PS_POSITIONED_DELETE • SQL_PS_POSITIONED_UPDATE • SQL_PS_SELECT_FOR_UPDATE, indicates whether or not the server requires the FOR UPDATE clause to be specified on a <query expression> in order for a column to be updateable using the cursor.
SQL_PROCEDURE_TERM	string	The name a database vendor uses for a procedure
SQL_PROCEDURES	string	A character string of "Y" indicates that the data source supports procedures and DB2 ODBC supports the ODBC procedure invocation syntax specified in "Using stored procedures" on page 417. "N" indicates that it does not.

Table 87. Information returned by SQLGetInfo (continued)

<i>InfoType</i>	Format	Description and notes
SQL_QUOTED_IDENTIFIER_CASE	16-bit integer	<p>Returns:</p> <ul style="list-style-type: none"> SQL_IC_UPPER - quoted identifiers in SQL are case insensitive and stored in upper case in the system catalog. SQL_IC_LOWER - quoted identifiers in SQL are case insensitive and are stored in lower case in the system catalog. SQL_IC_SENSITIVE - quoted identifiers (delimited identifiers) in SQL are case sensitive and are stored in mixed case in the system catalog. SQL_IC_MIXED - quoted identifiers in SQL are case insensitive and are stored in mixed case in the system catalog. <p>This should be contrasted with the SQL_IDENTIFIER_CASE <i>InfoType</i> which is used to determine how (unquoted) identifiers are stored in the system catalog.</p>
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 <i>InfoType</i> was SQL_OWNER_USAGE .)	32-bit mask	<p>Indicates the type of SQL statements that have schema (owners) associated with them when these statements are executed. Schema qualifiers (owners) are:</p> <ul style="list-style-type: none"> SQL_OU_DML_STATEMENTS - supported in all DML statements. SQL_OU_PROCEDURE_INVOCATION - supported in the procedure invocation statement. SQL_OU_TABLE_DEFINITION - supported in all table definition statements. SQL_OU_INDEX_DEFINITION - supported in all index definition statements. SQL_OU_PRIVILEGE_DEFINITION - supported in all privilege definition statements (i.e. grant and revoke statements).
SQL_SCROLL_CONCURRENCY	32-bit mask	<p>Indicates the concurrency options supported for the cursor.</p> <p>The following bit-masks are used in conjunction with the flag to determine which options are supported:</p> <ul style="list-style-type: none"> SQL_SCCO_READ_ONLY SQL_SCCO_LOCK SQL_SCCO_OPT_TIMESTAMP SQL_SCCO_OPT_VALUES <p>DB2 ODBC returns SQL_SCCO_LOCK indicating that the lowest level of locking that is sufficient to ensure the row can be updated is used.</p>
SQL_SCROLL_OPTIONS	32-bit mask	<p>The scroll options supported for scrollable cursors.</p> <p>The following bit-masks are used in conjunction with the flag to determine which options are supported:</p> <ul style="list-style-type: none"> SQL_SO_FORWARD_ONLY SQL_SO_KEYSET_DRIVEN SQL_SO_STATIC SQL_SO_DYNAMIC SQL_SO_MIXED <p>DB2 ODBC returns SQL_SO_FORWARD_ONLY, indicating that the cursor scrolls forward only.</p>

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_SEARCH_PATTERN_ESCAPE	string	Used to specify what the driver supports as an escape character for catalog functions such as (SQLTables(), SQLColumns()).
SQL_SERVER_NAME	string	The name of DB2 subsystem to which the application is connected.
SQL_SPECIAL_CHARACTERS	string	Contains all the characters in addition to a...z, A...Z, 0...9, and _ that the server allows in non-delimited identifiers.
SQL_SQL92_PREDICATES	32-bit mask	Indicates the predicates supported in a SELECT statement that SQL-92 defines. <ul style="list-style-type: none">• SQL_SP_BETWEEN• SQL_SP_COMPARISON• SQL_SP_EXISTS• SQL_SP_IN• SQL_SP_ISNOTNULL• SQL_SP_ISNULL• SQL_SP_LIKE• SQL_SP_MATCH_FULL• SQL_SP_MATCH_PARTIAL• SQL_SP_MATCH_UNIQUE_FULL• SQL_SP_MATCH_UNIQUE_PARTIAL• SQL_SP_OVERLAPS• SQL_SP_QUANTIFIED_COMPARISON• SQL_SP_UNIQUE
SQL_SQL92_VALUE_EXPRESSIONS	32-bit mask	Indicates the value expressions supported that SQL-92 defines. <ul style="list-style-type: none">• SQL_SVE_CASE• SQL_SVE_CAST• SQL_SVE_COALESCE• SQL_SVE_NULLIF
SQL_STATIC_SENSITIVITY	32-bit mask	Indicates whether changes made by an application with a positioned UPDATE or DELETE statement can be detected by that application: <ul style="list-style-type: none">• SQL_SS_ADDITIONS: Added rows are visible to the cursor; the cursor can scroll to these rows. All DB2 servers see added rows.• SQL_SS_DELETIONS: Deleted rows are no longer available to the cursor and do not leave a hole in the result set; after the cursor scrolls from a deleted row, it cannot return to that row.• SQL_SS_UPDATES: Updates to rows are visible to the cursor; if the cursor scrolls from and returns to an updated row, the data returned by the cursor is the updated data, not the original data.

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_STRING_FUNCTIONS	32-bit mask	<p data-bbox="911 247 1349 268">Indicates which string functions are supported.</p> <p data-bbox="911 300 1409 348">The following bit-masks are used to determine which string functions are supported:</p> <ul data-bbox="911 352 1219 848" style="list-style-type: none"> <li data-bbox="911 352 1133 373">• SQL_FN_STR_ASCII <li data-bbox="911 378 1138 399">• SQL_FN_STR_CHAR <li data-bbox="911 403 1166 424">• SQL_FN_STR_CONCAT <li data-bbox="911 428 1214 449">• SQL_FN_STR_DIFFERENCE <li data-bbox="911 453 1154 474">• SQL_FN_STR_INSERT <li data-bbox="911 478 1149 499">• SQL_FN_STR_LCASE <li data-bbox="911 504 1133 525">• SQL_FN_STR_LEFT <li data-bbox="911 529 1166 550">• SQL_FN_STR_LENGTH <li data-bbox="911 554 1166 575">• SQL_FN_STR_LOCATE <li data-bbox="911 579 1187 600">• SQL_FN_STR_LOCATE_2 <li data-bbox="911 604 1138 625">• SQL_FN_STR_LTRIM <li data-bbox="911 630 1166 651">• SQL_FN_STR_REPEAT <li data-bbox="911 655 1182 676">• SQL_FN_STR_REPLACE <li data-bbox="911 680 1149 701">• SQL_FN_STR_RIGHT <li data-bbox="911 705 1149 726">• SQL_FN_STR_RTRIM <li data-bbox="911 730 1187 751">• SQL_FN_STR_SOUNDEX <li data-bbox="911 756 1149 777">• SQL_FN_STR_SPACE <li data-bbox="911 781 1203 802">• SQL_FN_STR_SUBSTRING <li data-bbox="911 806 1154 827">• SQL_FN_STR_UCASE <p data-bbox="911 877 1458 1058">If an application can call the LOCATE scalar function with the <i>string1</i>, <i>string2</i>, and <i>start</i> arguments, the SQL_FN_STR_LOCATE bitmask is returned. If an application can only call the LOCATE scalar function with the <i>string1</i> and <i>string2</i>, the SQL_FN_STR_LOCATE_2 bitmask is returned. If the LOCATE scalar function is fully supported, both bitmasks are returned.</p>
SQL_SUBQUERIES	32-bit mask	<p data-bbox="911 1073 1349 1094">Indicates which predicates support subqueries:</p> <ul data-bbox="911 1098 1458 1253" style="list-style-type: none"> <li data-bbox="911 1098 1442 1119">• SQL_SQ_COMPARISON - the <i>comparison</i> predicate <li data-bbox="911 1123 1458 1144">• SQL_SQ_CORRELATE_SUBQUERIES - all predicates <li data-bbox="911 1148 1312 1169">• SQL_SQ_EXISTS - the <i>exists</i> predicate <li data-bbox="911 1173 1219 1194">• SQL_SQ_IN - the <i>in</i> predicate <li data-bbox="911 1199 1442 1253">• SQL_SQ_QUANTIFIED - the predicates containing a quantification scalar function.
SQL_SYSTEM_FUNCTIONS	32-bit mask	<p data-bbox="911 1268 1430 1289">Indicates which scalar system functions are supported.</p> <p data-bbox="911 1320 1409 1369">The following bit-masks are used to determine which scalar system functions are supported:</p> <ul data-bbox="911 1373 1203 1449" style="list-style-type: none"> <li data-bbox="911 1373 1170 1394">• SQL_FN_SYS_DBNAME <li data-bbox="911 1398 1154 1419">• SQL_FN_SYS_IFNULL <li data-bbox="911 1423 1203 1444">• SQL_FN_SYS_USERNAME <p data-bbox="911 1470 1458 1520">Note: These functions are intended to be used with the escape sequence in ODBC.</p>
SQL_TABLE_TERM	string	The database vendor's terminology for a table.

SQLGetInfo

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_TIMEDATE_ADD_INTERVALS	32-bit mask	<p>Indicates whether or not the special ODBC system function <code>TIMESTAMPADD</code> is supported, and, if it is, which intervals are supported.</p> <p>The following bitmasks are used to determine which intervals are supported:</p> <ul style="list-style-type: none"> • <code>SQL_FN_TSI_FRAC_SECOND</code> • <code>SQL_FN_TSI_SECOND</code> • <code>SQL_FN_TSI_MINUTE</code> • <code>SQL_FN_TSI_HOUR</code> • <code>SQL_FN_TSI_DAY</code> • <code>SQL_FN_TSI_WEEK</code> • <code>SQL_FN_TSI_MONTH</code> • <code>SQL_FN_TSI_QUARTER</code> • <code>SQL_FN_TSI_YEAR</code>
SQL_TIMEDATE_DIFF_INTERVALS	32-bit mask	<p>Indicates whether or not the special ODBC system function <code>TIMESTAMPDIFF</code> is supported, and, if it is, which intervals are supported.</p> <p>The following bitmasks are used to determine which intervals are supported:</p> <ul style="list-style-type: none"> • <code>SQL_FN_TSI_FRAC_SECOND</code> • <code>SQL_FN_TSI_SECOND</code> • <code>SQL_FN_TSI_MINUTE</code> • <code>SQL_FN_TSI_HOUR</code> • <code>SQL_FN_TSI_DAY</code> • <code>SQL_FN_TSI_WEEK</code> • <code>SQL_FN_TSI_MONTH</code> • <code>SQL_FN_TSI_QUARTER</code> • <code>SQL_FN_TSI_YEAR</code>
SQL_TIMEDATE_FUNCTIONS	32-bit mask	<p>Indicates which time and date functions are supported.</p> <p>The following bit-masks are used to determine which date functions are supported:</p> <ul style="list-style-type: none"> • <code>SQL_FN_TD_CURDATE</code> • <code>SQL_FN_TD_CURTIME</code> • <code>SQL_FN_TD_DAYNAME</code> • <code>SQL_FN_TD_DAYOFMONTH</code> • <code>SQL_FN_TD_DAYOFWEEK</code> • <code>SQL_FN_TD_DAYOFYEAR</code> • <code>SQL_FN_TD_HOUR</code> • <code>SQL_FN_TD_JULIAN_DAY</code> • <code>SQL_FN_TD_MINUTE</code> • <code>SQL_FN_TD_MONTH</code> • <code>SQL_FN_TD_MONTHNAME</code> • <code>SQL_FN_TD_NOW</code> • <code>SQL_FN_TD_QUARTER</code> • <code>SQL_FN_TD_SECOND</code> • <code>SQL_FN_TD_SECONDS_SINCE_MIDNIGHT</code> • <code>SQL_FN_TD_TIMESTAMPADD</code> • <code>SQL_FN_TD_TIMESTAMPDIFF</code> • <code>SQL_FN_TD_WEEK</code> • <code>SQL_FN_TD_YEAR</code> <p>Note: These functions are intended to be used with the escape sequence in ODBC.</p>

Table 87. Information returned by SQLGetInfo (continued)

InfoType	Format	Description and notes
SQL_TXN_CAPABLE	16-bit integer	Indicates whether transactions can contain DDL or DML or both. <ul style="list-style-type: none"> SQL_TC_NONE = transactions not supported. SQL_TC_DML = transactions can only contain DML statements (SELECT, INSERT, UPDATE, DELETE, etc.) DDL statements (CREATE TABLE, DROP INDEX, etc.) encountered in a transaction cause an error. SQL_TC_DDL_COMMIT = transactions can only contain DML statements. DDL statements encountered in a transaction cause the transaction to be committed. SQL_TC_DDL_IGNORE = transactions can only contain DML statements. DDL statements encountered in a transaction are ignored. SQL_TC_ALL = transactions can contain DDL and DML statements in any order.
SQL_TXN_ISOLATION_OPTION	32-bit mask	The transaction isolation levels available at the currently connected database server. <p>The following masks are used in conjunction with the flag to determine which options are supported:</p> <ul style="list-style-type: none"> SQL_TXN_READ_UNCOMMITTED SQL_TXN_READ_COMMITTED SQL_TXN_REPEATABLE_READ SQL_TXN_SERIALIZABLE SQL_TXN_NOCOMMIT SQL_TXN_VERSIONING <p>For descriptions of each level, see SQL_DEFAULT_TXN_ISOLATION.</p>
SQL_UNION	32-bit mask	Indicates if the server supports the UNION operator: <ul style="list-style-type: none"> SQL_U_UNION - supports the UNION clause SQL_U_UNION_ALL - supports the ALL keyword in the UNION clause <p>If SQL_U_UNION_ALL is set, so is SQL_U_UNION.</p>
SQL_USER_NAME	string	The user name used in a particular database. This is the identifier specified on the SQLConnect() call.
SQL_XOPEN_CLI_YEAR	string	Indicates the year of publication of the X/Open specification with which the version of the driver fully complies.

Table 88. 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

SQLGetInfo

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 89. SQLGetInfo SQLSTATES

SQLSTATE	Description	Explanation
01004	Data truncated.	The requested information is returned as a string and its length exceeds the length of the application buffer as specified in <i>BufferLength</i> . The argument <i>StringLengthPtr</i> contains the actual (not truncated) length of the requested information. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The type of information requested in <i>InfoType</i> requires an open connection. Only SQL_ODBC_VER does not require an open connection.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument <i>BufferLength</i> is less than 0.
HY096	Invalid information type.	An invalid <i>InfoType</i> was specified.
HYC00	Driver not capable.	The value specified in the argument <i>InfoType</i> is not supported by either DB2 ODBC or the data source.

Restrictions

None.

Example

```
SQLCHAR      buffer[255];
SQLSMALLINT  outlen;

rc = SQLGetInfo(hdbc, SQL_DATA_SOURCE_NAME, buffer, 255, &outlen);
printf("\nServer Name: %s\n", buffer);
```

References

- “SQLGetTypeInfo - Get data type information” on page 278
- “SQLGetConnectAttr - Get current attribute setting” on page 199

SQLGetLength - Retrieve length of a string value

Purpose

Specification:			
-----------------------	--	--	--

SQLGetLength() retrieves the length of a large object value, referenced by a large object locator that is returned from the server (as a result of a fetch, or an SQLGetSubString() call) during the current transaction.

Syntax

```
SQLRETURN SQLGetLength (SQLHSTMT hstmt,
                        SQLSMALLINT LocatorCType,
                        SQLINTEGER Locator,
                        SQLINTEGER FAR *StringLength,
                        SQLINTEGER FAR *IndicatorValue);
```

Function arguments

Table 90. SQLGetLength arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	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	The C type of the source LOB locator. This can be: <ul style="list-style-type: none"> SQL_C_BLOB_LOCATOR SQL_C_CLOB_LOCATOR SQL_C_DBCLOB_LOCATOR
SQLINTEGER	Locator	input	Must be set to the LOB locator value.
SQLINTEGER *	StringLength	output	The length of the returned information in <i>rgbValue</i> in bytes ^a if the target C buffer type is intended for a binary or character string variable and not a locator value. If the pointer is set to NULL, nothing is returned.
SQLINTEGER *	IndicatorValue	output	Always set to zero.

Note:

^a This is in bytes even for DBCLOB data.

Usage

SQLGetLength() can determine the length of the data value represented by a LOB locator. Applications use it to determine the overall length of the referenced LOB value so that the appropriate strategy for obtaining some or all of that value can be chosen.

The *Locator* argument can contain any valid LOB locator that is not explicitly freed using a FREE LOCATOR statement or 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.

SQLGetLength

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 91. SQLGetLength SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The combination of <i>LocatorCType</i> and <i>Locator</i> is not valid.
0F001	The LOB token variable does not currently represent any value.	The value specified for <i>Locator</i> is not associated with a LOB locator.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	<i>LocatorCType</i> is not one of the following: <ul style="list-style-type: none">• SQL_C_CLOB_LOCATOR• SQL_C_BLOB_LOCATOR• SQL_C_DBCLOB_LOCATOR
HY009	Invalid use of a null pointer.	Pointer to <i>StringLength</i> is NULL.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.

Restrictions

This function is not available when connected to a DB2 server that does not support large objects. Call `SQLGetFunctions()` with the function type set to `SQL_API_SQLGETLENGTH` and check the *fExists* output argument to determine if the function is supported for the current connection.

Example

See “Example” on page 261.

References

- “SQLBindCol - Bind a column to an application variable” on page 85
- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169
- “SQLFetch - Fetch next row” on page 176
- “SQLGetPosition - Return starting position of string” on page 259
- “SQLGetSubString - Retrieve portion of a string value” on page 275

SQLGetPosition - Return starting position of string

Purpose

Specification:			
-----------------------	--	--	--

SQLGetPosition() returns the starting position of one string within a LOB value (the source). The source value must be a LOB locator; the search string can be a LOB locator or a literal string.

The source and search LOB locators can be any value that is returned from the database from a fetch or a SQLGetSubString() call during the current transaction.

Syntax

```
SQLRETURN SQLGetPosition (SQLHSTMT          hstmt,
                          SQLSMALLINT       LocatorCType,
                          SQLINTEGER         SourceLocator,
                          SQLINTEGER         SearchLocator,
                          SQLCHAR           FAR *SearchLiteral,
                          SQLINTEGER         SearchLiteralLength,
                          SQLUINTEGER        FromPosition,
                          SQLUINTEGER FAR   *LocatedAt,
                          SQLINTEGER FAR    *IndicatorValue);
```

Function arguments

Table 92. SQLGetPosition arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	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	The C type of the source LOB locator. This can be: <ul style="list-style-type: none"> SQL_C_BLOB_LOCATOR SQL_C_CLOB_LOCATOR SQL_C_DBCLOB_LOCATOR
SQLINTEGER	Locator	input	<i>Locator</i> must be set to the source LOB locator.
SQLINTEGER	SearchLocator	input	If the <i>SearchLiteral</i> pointer is NULL and if <i>SearchLiteralLength</i> is set to 0, then <i>SearchLocator</i> must be set to the LOB locator associated with the search string. Otherwise, this argument is ignored.
SQLCHAR *	SearchLiteral	input	This argument points to the area of storage that contains the search string literal. If <i>SearchLiteralLength</i> is 0, this pointer must be NULL.
SQLINTEGER	SearchLiteralLength	input	The length of the string in <i>SearchLiteral</i> (in bytes). ^a If this argument value is 0, then the argument <i>SearchLocator</i> is meaningful.
SQLUINTEGER	FromPosition	input	For BLOBs and CLOBs, this is the position of the first byte within the source string at which the search is to start. For DBCLOBs, this is the first character. The start byte or character is numbered 1.

SQLGetPosition

Table 92. SQLGetPosition arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER *	LocatedAt	output	For BLOBs and CLOBs, this is the byte position at which the string was located or, if not located, the value zero. For DBCLOBs, this is the character position. If the length of the source string is zero, the value 1 is returned.
SQLINTEGER *	IndicatorValue	output	Always set to zero.

Note:

^a This is in bytes even for DBCLOB data.

Usage

SQLGetPosition() is used in conjunction with SQLGetSubString() to obtain any portion of a string in a random manner. To use SQLGetSubString(), the location of the substring within the overall string must be known in advance. In situations where a search string finds the start of that substring, SQLGetPosition() can be used to obtain the starting position of the substring.

The *Locator* and *SearchLocator* arguments (if used) can contain any valid LOB locator that is not explicitly freed using a FREE LOCATOR statement or implicitly freed because the transaction during which it was created has terminated.

The *Locator* and *SearchLocator* must have the same LOB locator type.

The statement handle must not be associated with any prepared statements or catalog function calls.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 93. SQLGetPosition SQLSTATES

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The combination of <i>LocatorCType</i> and either of the LOB locator values is not valid.
0F001	The LOB token variable does not currently represent any value.	The value specified for <i>Locator</i> or <i>SearchLocator</i> is not currently a LOB locator.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
42818	The operands of an operator or function are not compatible.	The length of the pattern is longer than 4000 bytes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.

Table 93. SQLGetPosition SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY009	Invalid use of a null pointer.	The pointer to the <i>LocatedAt</i> argument is NULL. The argument value for <i>FromPosition</i> is not greater than 0. <i>LocatorCType</i> is not one of SQL_C_CLOB_LOCATOR, SQL_C_BLOB_LOCATOR, or SQL_C_DBCLOB_LOCATOR.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value of <i>SearchLiteralLength</i> is less than 1, and not SQL_NTS.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.

Restrictions

This function is not available when connected to a DB2 server that does not support large objects. Call `SQLGetFunctions()` with the function type set to `SQL_API_SQLGETPOSITION` and check the *fExists* output argument to determine if the function is supported for the current connection.

Example

```

/* ... */
SQLCHAR          stmt2[] =
                  "SELECT resume FROM emp_resume "
                  "WHERE empno = ? AND resume_format = 'ascii';
/* ... */
/*****
** Get CLOB locator to selected Resume **
*****/
rc = SQLSetParam(hstmt, 1, SQL_C_CHAR, SQL_CHAR, 7,
                 0, 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);

```

SQLGetPosition

```
/*
*****
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 #: %s\n %s\n", Empno.s, buffer);
/* ... */
```

References

- “SQLBindCol - Bind a column to an application variable” on page 85
- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169
- “SQLFetch - Fetch next row” on page 176
- “SQLGetFunctions - Get functions” on page 228
- “SQLGetLength - Retrieve length of a string value” on page 257
- “SQLGetSubString - Retrieve portion of a string value” on page 275

SQLGetSQLCA - Get SQLCA data structure

Purpose

Specification:			
-----------------------	--	--	--

SQLGetSQLCA() is used to return the SQLCA associated with preparing and executing an SQL statement, fetching data, or closing a cursor. The SQLCA can return information that supplements the information obtained by using SQLGetDiagRec().

For a detailed description of the SQLCA structure, see Appendix C of *DB2 SQL Reference*.

An SQLCA is not available if a function is processed strictly on the application side, such as allocating a statement handle. In this case, an empty SQLCA is returned with all values set to zero.

Syntax

```
SQLRETURN SQLGetSQLCA(
    (SQLHENV      henv,
     SQLHDBC      hdbc,
     SQLHSTMT     hstmt,
     struct sqlca FAR *pSqlca );
```

Function arguments

Table 94. SQLGetSQLCA arguments

Data type	Argument	Use	Description
SQLHENV	henv	input	Environment Handle
SQLHDBC	hdbc	input	Connection Handle
SQLHSTMT	hstmt	input	Statement Handle
SQLCA *	pqlca	output	SQL Communication Area

Usage

The handles are used in the same way as for the SQLGetDiagRec() function. To obtain the SQLCA associated with:

- An environment, pass a valid environment handle. Set *hdbc* and *hstmt* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
- A connection, pass a valid database connection handle, and set *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored.
- A statement, pass a valid statement handle. The *henv* and *hdbc* arguments are ignored.

If diagnostic information generated by one DB2 ODBC function is not retrieved before a function other than SQLGetDiagRec() is called with the same handle, the information for the previous function call is lost. This is true whether or not diagnostic information is generated for the second DB2 ODBC function call.

SQLGetSQLCA

If a DB2 ODBC function is called that does not result in interaction with the DBMS, then the SQLCA contains all zeroes. Meaningful information is returned for the following functions:

- SQLCancel()
- SQLConnect(), SQLDisconnect()
- SQLExecDirect(), SQLExecute()
- SQLFetch()
- SQLPrepare()
- SQLEndTran()
- SQLColumns()
- SQLConnect()
- SQLGetData (if LOB column is involved)
- SQLSetConnectAttr() (for SQL_AUTOCOMMIT)
- SQLStatistics()
- SQLTables()
- SQLColumnPrivileges()
- SQLExtendedFetch()
- SQLForeignKeys()
- SQLMoreResults()
- SQLPrimaryKeys()
- SQLProcedureColumns()
- SQLProcedures()
- SQLTablePrivileges()

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

None.

Restrictions

None.

Example

```

/*****
/*      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, i.e. this 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;

```

SQLGetSQLCA

```

/*****
/* 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,
                empToyee,
                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 = %d.\n", rc);
    goto exit;
}

code = (int) pSQLCA->sqlcode;
memcpy (state, pSQLCA->sqlstate, 5);
state [5] = '\0';

(void) printf ("**** sqlcode = %d, sqlstate = %s.\n", code, state);

memcpy (errp, pSQLCA->sqlerrp, 8);
errp [8] = '\0';
(void) printf ("**** sqlerrp = %s.\n", errp);

if (pSQLCA->sqlerrml == 0)
    (void) printf ("**** No tokens.\n");
else
{
    for (len = 0, count = 0; len < pSQLCA->sqlerrml; len = ++end)
    {
        start = end = len;
        while ((pSQLCA->sqlerrmc [end] != 0xFF) &&
              (end < pSQLCA->sqlerrml))

            end++;
        if (start != end)
        {
            memcpy (tok, &pSQLCA->sqlerrmc[start],
                  (end-start));

            tok [end-start+1] = '\0';
            (void) printf ("**** Token # %d = %s.\n", count++, tok);
        }
    }
}

for (i = 0; i <= 5; i++)
    (void) printf ("**** sqlerrd # %d = %d.\n", i+1, pSQLCA->sqlerrd_i);

for (i = 0; i <= 10; i++)
    (void) printf ("**** sqwarn # %d = %c.\n", i+1, pSQLCA->sqlwarn_i);

exit:
return;
}

```

References

- “SQLGetDiagRec - Get multiple field settings of diagnostic record” on page 223

SQLGetStmtAttr - Get current setting of a statement attribute

Purpose

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLGetStmtAttr() returns the current setting of a statement attribute. These options are set using the SQLSetStmtAttr() function.

Syntax

```
SQLRETURN SQLGetStmtAttr (SQLHSTMT      StatementHandle,
                          SQLINTEGER     Attribute,
                          SQLPOINTER     ValuePtr,
                          SQLINTEGER     BufferLength,
                          SQLINTEGER     *StringLengthPtr);
```

Function arguments

Table 95. SQLGetStmtAttr arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>StatementHandle</i>	input	Connection handle.
SQLINTEGER	<i>Attribute</i>	input	Statement attribute to retrieve. Refer to Table 150 on page 361 for a complete list of attributes.
SQLPOINTER	<i>ValuePtr</i>	output	A pointer to memory in which to return the current value of the attribute specified by <i>Attribute</i> . <i>*ValuePtr</i> will be a 32-bit unsigned integer value or point to a null-terminated character string. If the <i>Attribute</i> argument is a driver-specific value, the value in <i>*ValuePtr</i> may be a signed integer.
SQLINTEGER	<i>BufferLength</i>	input	<ul style="list-style-type: none"> For ODBC-defined attributes: <ul style="list-style-type: none"> If <i>ValuePtr</i> points to a character string, this argument should be the length of <i>*ValuePtr</i>. If <i>ValuePtr</i> points to an integer, <i>BufferLength</i> is ignored. For driver-defined attributes (IBM extension): <ul style="list-style-type: none"> If <i>ValuePtr</i> points to a character string, this argument should be the length of <i>*ValuePtr</i> or SQL_NTS. If SQL_NTS, the driver assumes the length of <i>*ValuePtr</i> to be SQL_MAX_OPTIONS_STRING_LENGTH bytes (excluding the null-terminator). If <i>ValuePtr</i> points to an integer, <i>BufferLength</i> is ignored.
SQLINTEGER *	<i>StringLengthPtr</i>	output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes returned for the null-termination character) available to return in <i>ValuePtr</i> . <ul style="list-style-type: none"> If <i>ValuePtr</i> 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 <i>BufferLength</i>, the data in <i>ValuePtr</i> is truncated to <i>BufferLength</i> minus the length of a null-termination character and is null-terminated by DB2 ODBC. If <i>Attribute</i> does not denote a string, DB2 ODBC ignores <i>BufferLength</i> and does not set <i>StringLengthPtr</i>.

Usage

SQLGetStmtAttr() returns the current setting of a statement attribute. These options are set using the SQLSetStmtAttr() function. For a list of valid environment attributes, refer to Table 150 on page 361.

For information about overriding DB2 CCSIDs from DSNHDECP, see “Usage” on page 360.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

Table 96. SQLGetStmtAttr SQLSTATES

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The data returned in *ValuePtr was truncated to be BufferLength minus the length of a null termination character. The length of the untruncated string value is returned in *StringLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state.	The argument Attribute was SQL_ATTR_ROW_NUMBER and the cursor was not open, or the cursor was positioned before the start of the result set or after the end of the result set.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by SQLGetDiagRec() in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC was not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY013	Unexpected memory handling error.	DB2 ODBC was not able to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument BufferLength was less than 0.
HY092	Option type out of range.	The value specified for the argument Attribute was not valid for this version of DB2 ODBC.
HY109	Invalid cursor position.	The Attribute argument was SQL_ATTR_ROW_NUMBER and the row had been deleted or could not be fetched.
HYC00	Driver not capable.	The value specified for the argument Attribute was a valid connection or statement attribute for the version of the DB2 ODBC driver, but was not supported by the data source.

Restrictions

None.

SQLGetStmtAttr

Example

```
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" );
```

References

- “SQLSetStmtAttr - Set options related to a statement” on page 360
- “SQLSetConnectAttr - Set connection attributes” on page 336
- “SQLGetConnectAttr - Get current attribute setting” on page 199

SQLGetStmtOption - Returns current setting of a statement option

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	
-----------------------	-----------------	-------------------	--

In ODBC 3.0, SQLGetStmtAttr() replaces the ODBC 2.0 function SQLGetStmtOption(). See SQLGetStmtAttr() for more information.

SQLGetStmtOption() returns the current settings of the specified statement option.

These options are set using the SQLSetStmtOption() function.

Syntax

```
SQLRETURN SQLGetStmtOption (SQLHSTMT      hstmt,
                             SQLUSMALLINT  fOption,
                             SQLPOINTER    pvParam);
```

Function arguments

Table 97. SQLGetStmtOption arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLUSMALLINT	fOption	input	Option to set.
SQLPOINTER	pvParam	output	Value of the option. Depending on the value of <i>fOption</i> this can be a 32-bit integer value, or a pointer to a null terminated character string. The maximum length of any character string returned is SQL_MAX_OPTION_STRING_LENGTH bytes (excluding the null-terminator).

Usage

See Table 150 on page 361 in the function description of SQLSetStmtAttr() for a list of statement options. The following table lists the statement options that are read-only (can be read but not set).

Table 98. Statement options

<i>fOption</i>	Contents
SQL_ROW_NUMBER	A 32-bit integer value that specifies 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, 0 is returned.

Note: ODBC architecture also defines the read-only statement option SQL_GET_BOOKMARK. This option is not supported by DB2 ODBC. If it is specified, this function returns SQL_ERROR (SQLSTATE HY011 -- Operation invalid at this time.)

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

SQLGetStmtOption

Diagnostics

Table 99. SQLGetStmtOption SQLSTATES

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	There is no open cursor on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid use of a null pointer.	<i>pvParam</i> was null.
S1010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
S1092	Option type out of range.	An invalid <i>fOption</i> value was specified.
S1C00	Driver not capable.	DB2 ODBC recognizes the option but does not support it.

Restrictions

None.

Example

```
/* ... */
rc = SQLGetStmtOption(hstmt, SQL_CURSOR_HOLD, &cursor_hold);
printf("Cursor With Hold is: ");
if (cursor_hold == SQL_CURSOR_HOLD_ON )
    printf("ON\n");
else
    printf("OFF\n");
/* ... */
```

References

- “SQLSetConnectOption - Set connection option” on page 345
- “SQLSetStmtOption - Set statement option” on page 367

SQLGetSubString - Retrieve portion of a string value

Purpose

Specification:			
-----------------------	--	--	--

SQLGetSubString() retrieves a portion of a large object value, referenced by a LOB locator that the server returns (returned by a fetch or a previous SQLGetSubString() call) during the current transaction.

Syntax

```
SQLRETURN SQLGetSubString (SQLHSTMT      hstmt,
                          SQLSMALLINT    LocatorCType,
                          SQLINTEGER      SourceLocator,
                          SQLUINTEGER     FromPosition,
                          SQLUINTEGER     ForLength,
                          SQLSMALLINT     TargetCType,
                          SQLPOINTER      rgbValue,
                          SQLINTEGER      cbValueMax,
                          SQLINTEGER FAR  *StringLength,
                          SQLINTEGER FAR  *IndicatorValue);
```

Function arguments

Table 100. SQLGetSubString arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	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	The C type of the source LOB locator. This can be: <ul style="list-style-type: none"> SQL_C_BLOB_LOCATOR SQL_C_CLOB_LOCATOR SQL_C_DBCLOB_LOCATOR
SQLINTEGER	Locator	input	<i>Locator</i> must be set to the source LOB locator value.
SQLUINTEGER	FromPosition	input	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 character is numbered 1.
SQLUINTEGER	ForLength	input	This is the length of the string to be returned by the function. For BLOBs and CLOBs, this is the length in bytes. For DBCLOBs, this is the length in characters. <p>If <i>FromPosition</i> is less than the length of the source string but <i>FromPosition</i> + <i>ForLength</i> - 1 extends beyond the end of the source string, the result is padded on the right with the necessary number of characters (X'00' for BLOBs, single-byte blank character for CLOBs, and double-byte blank character for DBCLOBs).</p>
SQLSMALLINT	TargetCType	input	The C data type of the <i>rgbValue</i> . The target must always be either a LOB locator C buffer type (SQL_C_CLOB_LOCATOR, SQL_C_BLOB_LOCATOR, SQL_C_DBCLOB_LOCATOR) or a C string variable (SQL_C_CHAR for CLOB, SQL_C_BINARY for BLOB, and SQL_C_DBCHAR for DBCLOB).
SQLPOINTER	rgbValue	output	Pointer to the buffer where the retrieved string value or a LOB locator is stored.

SQLGetSubString

Table 100. SQLGetSubString arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER	cbValueMax	input	Maximum size of the buffer pointed to by <i>rgbValue</i> in bytes.
SQLINTEGER *	StringLength	output	The length of the returned information in <i>rgbValue</i> in bytes ^a if the target C buffer type is intended for a binary or character string variable and not a locator value. If the pointer is set to NULL, nothing is returned.
SQLINTEGER *	IndicatorValue	output	Always set to zero.

Note:

^a This is in bytes even for DBCLOB data.

Usage

SQLGetSubString() is used to obtain any portion of the string that is represented by the LOB locator. The target can be one of the following:

- 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.

SQLGetSubString() can be used as an alternative to SQLGetData() for getting data in pieces. In this case, a column is first bound to a LOB locator, which is then used to fetch the LOB as a whole or in pieces.

The *Locator* argument can contain any valid LOB locator that is not explicitly freed using a FREE LOCATOR statement or 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

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 101. SQLGetSubString SQLSTATES

SQLSTATE	Description	Explanation
01004	Data truncated.	The amount of returned data is longer than <i>cbValueMax</i> . Actual length available for return is stored in <i>StringLength</i> .
07006	Invalid conversion.	The value specified for <i>TargetCType</i> is not SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR or a LOB locator. The value specified for <i>TargetCType</i> is inappropriate for the source (for example SQL_C_DBCHAR for a BLOB column).
0F001	The LOB token variable does not currently represent any value.	The value specified for <i>Locator</i> or <i>SearchLocator</i> is not currently a LOB locator.
22011	A substringing error occurred.	<i>FromPosition</i> is greater than the length of the source string.

Table 101. SQLGetSubstring SQLSTATEs (continued)

SQLSTATE	Description	Explanation
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	<i>LocatorCType</i> is not one of the following: <ul style="list-style-type: none"> • SQL_C_CLOB_LOCATOR • SQL_C_BLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY024	Invalid argument value.	The value specified for <i>FromPosition</i> or for <i>ForLength</i> is not a positive integer.
HY090	Invalid string or buffer length.	The value of <i>cbValueMax</i> is less than 0.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.

Restrictions

This function is not available when connected to a DB2 server that does not support large objects. Call `SQLGetFunctions()` with the function type set to `SQL_API_SQLGETSUBSTRING`, and check the *fExists* output argument to determine if the function is supported for the current connection.

Example

See “Example” on page 261.

References

- “SQLBindCol - Bind a column to an application variable” on page 85
- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169
- “SQLFetch - Fetch next row” on page 176
- “SQLGetLength - Retrieve length of a string value” on page 257
- “SQLGetSubString - Retrieve portion of a string value” on page 275

SQLGetTypeInfo - Get data type information

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLGetTypeInfo() returns information about the data types that are supported by the DBMSs associated with DB2 ODBC. The information is returned in an SQL result set. The columns can be received using the same functions that are used to process a query.

Syntax

```
SQLRETURN SQLGetTypeInfo (SQLHSTMT          hstmt,
                          SQLSMALLINT       fSqlType);
```

Function arguments

Table 102. SQLGetTypeInfo arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle.
SQLSMALLINT	<i>fSqlType</i>	input	<p>The SQL data type being queried. The supported types are:</p> <ul style="list-style-type: none"> • SQL_ALL_TYPES • SQL_BINARY • SQL_BLOB • SQL_CHAR • SQL_CLOB • SQL_DBCLOB • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARIABLE • SQL_LONGVARIABLE • SQL_LONGVARIABLE • SQL_NUMERIC • SQL_REAL • SQL_ROWID • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARIABLE • SQL_VARIABLE • SQL_VARIABLE <p>If 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.</p>

Usage

Since SQLGetTypeInfo() generates a result set and is equivalent to executing a query, it generates a cursor and begins a transaction. To prepare and execute another statement on this statement handle, the cursor must be closed.

If `SQLGetTypeInfo()` is called with an invalid *fSqlType*, an empty result set is returned.

The columns of the result set generated by this function are described below.

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 data types returned are those that can be used in a CREATE TABLE, ALTER TABLE, DDL statement. Non-persistent data types such as the locator data types are not part of the returned result set. User defined data types are not returned either.

Table 103. Columns returned by `SQLGetTypeInfo`

Column number/name	Data type	Description
1 TYPE_NAME	VARCHAR(128) NOT NULL	Character representation of the SQL data type name. For example, VARCHAR, BLOB, DATE, INTEGER.
2 DATA_TYPE	SMALLINT NOT NULL	SQL data type define values. For example, SQL_VARCHAR, SQL_BLOB, SQL_TYPE_DATE, SQL_INTEGER.
3 COLUMN_SIZE	INTEGER	<p>If the data type is a character or binary string, then this column contains the maximum length in bytes; if it is a graphic (DBCS) string, this is the number of double byte characters for the column.</p> <p>For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character.</p> <p>For numeric data types, this is the total number of digits.</p>
4 LITERAL_PREFIX	VARCHAR(128)	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)	Character that DB2 recognizes as a suffix for a literal of this data type. This column is null for data types where a literal prefix is not applicable.
6 CREATE_PARAMS	VARCHAR(128)	<p>The text of this column contains a list of keywords, separated by commas, that correspond to each parameter the application can specify in parenthesis when using the name in the TYPE_NAME column as a data type in SQL. The keywords in the list can be any of the following: LENGTH, PRECISION, SCALE. They appear in the order that the SQL syntax requires that they be used.</p> <p>A NULL indicator is returned if there are no parameters for the data type definition, (such as INTEGER).</p> <p>Note: The intent of CREATE_PARAMS is to enable an application to customize the interface for a <i>DDL builder</i>. An application should expect, using this, only to be able to determine the number of arguments required to define the data type and to have localized text that could be used to label an edit control.</p>
7 NULLABLE	SMALLINT NOT NULL	<p>Indicates whether the data type accepts a NULL value</p> <ul style="list-style-type: none"> • Set to SQL_NO_NULLS if NULL values are disallowed. • Set to SQL_NULLABLE if NULL values are allowed.

SQLGetTypeInfo

Table 103. Columns returned by SQLGetTypeInfo (continued)

Column number/name	Data type	Description
8 CASE_SENSITIVE	SMALLINT NOT NULL	Indicates whether the data type can be treated as case sensitive for collation purposes; valid values are SQL_TRUE and SQL_FALSE.
9 SEARCHABLE	SMALLINT NOT NULL	Indicates how the data type is used in a WHERE clause. Valid values are: <ul style="list-style-type: none">• SQL_UNSEARCHABLE : if the data type cannot be used in a WHERE clause.• SQL_LIKE_ONLY : if the data type can be used in a WHERE clause only with the LIKE predicate.• SQL_ALL_EXCEPT_LIKE : if the data type can be used in a WHERE clause with all comparison operators except LIKE.• SQL_SEARCHABLE : if the data type can be used in a WHERE clause with any comparison operator.
10 UNSIGNED_ATTRIBUTE	SMALLINT	Indicates whether the data type is unsigned. The valid values are: SQL_TRUE, SQL_FALSE or NULL. A NULL indicator is returned if this attribute is not applicable to the data type.
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)	This column contains any localized (native language) name for the data type that is different from the regular name of the data type. If there is no localized name, this column is NULL. This column is intended for display only. The character set of the string is locale-dependent and is typically the default character set of the database.
14 MINIMUM_SCALE	SMALLINT	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.
15 MAXIMUM_SCALE	SMALLINT	The maximum scale of the SQL data type. NULL is returned where scale is not applicable. If the maximum scale is not defined separately in the DBMS, but is defined instead to be the same as the maximum length of the column, then this column contains the same value as the COLUMN_SIZE column.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 104. SQLGetTypeInfo SQLSTATES

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle. <i>hstmt</i> is not closed.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY004	Invalid SQL data type.	An invalid <i>fSqlType</i> is specified.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.

Restrictions

The following ODBC specified SQL data types (and their corresponding *fSqlType* define values) are not supported by any IBM RDBMS:

Data type	fSqlType
TINY INT	SQL_TINYINT
BIG INT	SQL_BIGINT
BIT	SQL_BIT

Example

```

/*****
/* Invokes 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;

```

SQLGetTypeInfo

```
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 null-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);

```

SQLGetTypeInfo

```
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);  
}
```

References

- “SQLColAttribute - Get column attributes” on page 106
- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169
- “SQLGetInfo - Get general information” on page 234

SQLMoreResults - Determine if there are more result sets

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLMoreResults() determines whether there is more information available on the statement handle which has been associated with:

- Array input of parameter values for a query, or
- A stored procedure that is returning result sets.

Syntax

```
SQLRETURN SQLMoreResults (SQLHSTMT hstmt);
```

Function arguments

Table 105. SQLMoreResults arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.

Usage

This function is used to return multiple results set in a sequential manner upon the execution of:

- A parameterized query with an array of input parameter values specified with SQLParamOptions() and SQLBindParameter(), or
- A stored procedure containing SQL queries, the cursors of which have been left open so that the result sets remain accessible when the stored procedure has finished execution.

See “Using arrays to input parameter values” on page 403 and “Returning result sets from stored procedures” on page 420 for more information.

After completely processing the first result set, the application can call SQLMoreResults() to determine if another result set is available. If the current result set has unfetched rows, SQLMoreResults() discards them by closing the cursor and, if another result set is available, returns SQL_SUCCESS.

If all the result sets have been processed, SQLMoreResults() returns SQL_NO_DATA_FOUND.

If SQLFreeStmt() is called with the SQL_CLOSE option or SQLFreeHandle() is called with *HandleType* set to SQL_HANDLE_STMT, all pending result sets on this statement handle are discarded.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

Diagnostics

Table 106. SQLMoreResults SQLSTATEs

SQLSTATE	Description	Explanation
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.

In addition `SQLMoreResults()` can return the SQLSTATEs associated with `SQLExecute()`.

Restrictions

The ODBC specification of `SQLMoreResults()` also allows counts associated with the execution of parameterized INSERT, UPDATE, and DELETE statements with arrays of input parameter values to be returned. However, DB2 ODBC does not support the return of such count information.

Example

```

/* ... */
#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 "
  ") "

```

SQLMoreResults

```
/* 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];

SQLINTEGER    pirow = 0;
SQLINTEGER    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)/ROW_SIZE, 0);

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

/* Bind remaining columns ... */
/* ... */
```

```

/* NOTE: This sample assumes that an order never has more
        rows than ROWSET_SIZE. A check should be added below to call
        SQLExtendedFetch multiple times for each result set.
*/
do /* for each result set .... */
{ rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, NULL);

  if (pcrow > 0) /* if 1 or more rows in the result set */
  {
    i = j = 0;

    printf("*****\n");
    printf("Orders for Customer: %ld\n", Ord[0].Cust_Num);
    printf("*****\n");

    while (i < pcrow)
    { printf("\nOrder #: %ld\n", Ord[i].Ord_Num);
      printf("      Product  Quantity      Price\n");
      printf("      -----  -----  -----");
      j = i;
      while (Ord[j].Ord_Num == Ord[i].Ord_Num)
      { printf("      %8ld %16.7lf %12.2lf\n",
                Ord[i].Prod_Num, Ord[i].Quant, Ord[i].Amount);
        i++;
      }
      printf("                                =====\n");
      printf("                                %12.2lf\n", Ord[j].Total);
    } /* end while */
  } /* end if */
}
while ( SQLMoreResults(hstmt) == SQL_SUCCESS);
/* ... */

```

References

- “SQLParamOptions - Specify an input array for a parameter” on page 298

SQLNativeSql - Get native SQL text

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLNativeSql() is used to show how DB2 ODBC interprets vendor escape clauses. If the original SQL string passed in by the application contains vendor escape clause sequences, then DB2 ODBC returns the transformed SQL string that the data source sees (with vendor escape clauses either converted or discarded, as appropriate).

Syntax

```
SQLRETURN SQLNativeSql (SQLHDBC hdbc,
                        SQLCHAR FAR *szSqlStrIn,
                        SQLINTEGER cbSqlStrIn,
                        SQLCHAR FAR *szSqlStr,
                        SQLINTEGER cbSqlStrMax,
                        SQLINTEGER FAR *pcbSqlStr);
```

Function arguments

Table 107. SQLNativeSQL arguments

Data type	Argument	Use	Description
SQLHDBC	hdbc	input	Connection handle.
SQLCHAR *	szSqlStrIn	input	Input SQL string.
SQLINTEGER	cbSqlStrIn	input	Length of <i>szSqlStrIn</i> .
SQLCHAR *	szSqlStr	output	Pointer to buffer for the transformed output string.
SQLINTEGER	cbSqlStrMax	input	Size of buffer pointed by <i>szSqlStr</i> .
SQLINTEGER *	pcbSqlStr	output	The total number of bytes (excluding the null-terminator) available to return in <i>szSqlStr</i> . If the number of bytes available to return is greater than or equal to <i>cbSqlStrMax</i> , the output SQL string in <i>szSqlStr</i> is truncated to <i>cbSqlStrMax - 1</i> bytes.

Usage

This function is called when the application wishes to examine or display the transformed SQL string that is passed to the data source by DB2 ODBC. Translation (mapping) only occurs if the input SQL statement string contains vendor escape clause sequences. For more information on vendor escape clause sequences, see “Using vendor escape clauses” on page 448.

DB2 ODBC can only detect vendor escape clause syntax errors; since DB2 ODBC does not pass the transformed SQL string to the data source for preparation, syntax errors that are detected by the DBMS are not generated 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

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR

- SQL_INVALID_HANDLE

Diagnostics

Table 108. SQLNativeSQL SQLSTATES

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer <i>szSqlStr</i> is not large enough to contain the entire SQL string, so truncation occurs. The argument <i>pcbSqlStr</i> contains the total length of the untruncated SQL string. (Function returns with SQL_SUCCESS_WITH_INFO)
08003	Connection is closed.	The <i>hdbc</i> does not reference an open database connection.
37000	Invalid SQL syntax.	The input SQL string in <i>szSqlStrIn</i> contained a syntax error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	The argument <i>szSqlStrIn</i> is a NULL pointer. The argument <i>szSqlStr</i> is a NULL pointer.
HY090	Invalid string or buffer length.	The argument <i>cbSqlStrIn</i> was less than 0, but not equal to SQL_NTS. The argument <i>cbSqlStrMax</i> was less than 0.

Restrictions

None.

Example

```

/* ... */
    SQLCHAR        in_stmt[1024];
    SQLCHAR        out_stmt[1024];
    SQLSMALLINT    pcPar;
    SQLINTEGER     indicator;
/* ... */
/* Prompt for a statement to prepare */
printf("Enter an SQL statement: \n");
gets(in_stmt);

/* prepare the statement */
rc = SQLPrepare(hstmt, in_stmt, SQL_NTS);

SQLNumParams(hstmt, &pcPar);

SQLNativeSql(hstmt, in_stmt, SQL_NTS, out_stmt, 1024, &indicator);

if (indicator == SQL_NULL_DATA)
{ printf("Invalid statement\n"); }
else
{ printf(" Input Statement: \n %s \n", in_stmt);
  printf("Output Statement: \n %s \n", out_stmt);
  printf("Number of Parameter Markers = %ld\n", pcPar);
}
rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
/* ... */

```

References

- “Using vendor escape clauses” on page 448

SQLNumParams - Get number of parameters in a SQL statement

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLNumParams() returns the number of parameter markers in a SQL statement.

Syntax

```
SQLRETURN SQLNumParams (SQLHSTMT hstmt,
                        SQLSMALLINT FAR *pcpar);
```

Function arguments

Table 109. SQLNumParams arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLSMALLINT *	pcpar	Output	Number of parameters in the statement.

Usage

This function can only be called after the statement associated with *hstmt* has been prepared. If the statement does not contain any parameter markers, *pcpar* is set to 0.

An application can call this function to determine how many SQLBindParameter() calls are necessary for the SQL statement associated with the statement handle.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 110. SQLNumParams SQLSTATES

SQLSTATE	Description	Explanation
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	<i>pcpar</i> is null.
HY010	Function sequence error.	This function is called before SQLPrepare() is called for the specified <i>hstmt</i> . The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

See “Example” on page 291.

References

- “SQLBindParameter - Binds a parameter marker to a buffer or LOB locator” on page 91
- “SQLPrepare - Prepare a statement” on page 300

SQLNumResultCols - Get number of result columns

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLNumResultCols() returns the number of columns in the result set associated with the input statement handle.

SQLPrepare() or SQLExecDirect() must be called before calling this function.

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

Syntax

```
SQLRETURN SQLNumResultCols (SQLHSTMT hstmt,
                             SQLSMALLINT FAR *pccol);
```

Function arguments

Table 111. SQLNumResultCols arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle
SQLSMALLINT *	<i>pccol</i>	output	Number of columns in the result set

Usage

The function sets the output argument to zero if the last statement or function executed on the input statement handle did not generate a result set.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 112. SQLNumResultCols SQLSTATES

SQLSTATE	Description	Explanation
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	<i>pccol</i> is a null pointer.
HY010	Function sequence error.	The function is called prior to calling SQLPrepare() or SQLExecDirect() for the <i>hstmt</i> . The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.

Table 112. SQLNumResultCols SQLSTATEs (continued)

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

Restrictions

None.

Example

See “Example” on page 139

References

- “SQLColAttribute - Get column attributes” on page 106
- “SQLDescribeCol - Describe column attributes” on page 137
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLGetData - Get data from a column” on page 210
- “SQLPrepare - Prepare a statement” on page 300

SQLParamData - Get next parameter for which a data value is needed

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLParamData() is used in conjunction with SQLPutData() to send long data in pieces. It can also be used to send fixed length data as well. For a description of the exact sequence of this input method, see “Sending/retrieving long data in pieces” on page 401.

Syntax

```
SQLRETURN SQLParamData (SQLHSTMT hstmt,
                        SQLPOINTER FAR *prgbValue);
```

Function arguments

Table 113. SQLParamData arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLPOINTER *	prgbValue	output	Pointer to the value of the <i>prgbValue</i> argument specified on the SQLBindParameter() or SQLSetParam() call.

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* supplied by the application during the previous SQLBindParameter() call. SQLPutData() is called one or more times (in the case of long data) to send the parameter data. SQLParamData() is called to signal that all the data has been sent for the current parameter and to advance to the next SQL_DATA_AT_EXEC parameter. SQL_SUCCESS is returned when all the parameters have been assigned data values and the associated statement has been executed successfully. If any errors occur during or before actual statement execution, SQL_ERROR is returned.

If SQLParamData() returns SQL_NEED_DATA, then only SQLPutData() or SQLCancel() calls can be made. All other function calls using this statement handle fails. 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()

Should they be invoked during an SQL_NEED_DATA sequence, these functions return SQL_ERROR with SQLSTATE of HY010 and the processing of the SQL_DATA_AT_EXEC parameters is not affected.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA

Diagnostics

SQLParamData() can return any SQLSTATE returned by the SQLExecDirect() and SQLExecute() functions. In addition, the following diagnostics can also be generated:

Table 114. SQLParamData SQLSTATES

SQLSTATE	Description	Explanation
40001	Transaction rollback.	The transaction to which this SQL statement belonged is rolled back due to a deadlock or timeout.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	SQLParamData() is called out of sequence. This call is only valid after an SQLExecDirect() or an SQLExecute(), or after an SQLPutData() call. Even though this function is called after an SQLExecDirect() or an SQLExecute() call, there are no SQL_DATA_AT_EXEC parameters (left) to process.

Restrictions

None.

Example

See “Example” on page 329.

References

- “SQLBindParameter - Binds a parameter marker to a buffer or LOB locator” on page 91
- “SQLCancel - Cancel statement” on page 102
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLPutData - Passing data value for a parameter” on page 327
- “SQLSetParam - Binds a parameter marker to a buffer” on page 354

SQLParamOptions - Specify an input array for a parameter

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLParamOptions() provides the ability to set multiple values for each parameter set by SQLBindParameter(). This allows the application to perform batched processing of the same SQL statement with one set of prepare, execute and SQLBindParameter() calls.

Syntax

```
SQLRETURN SQLParamOptions (SQLHSTMT hstmt,
                            SQLINTEGER crow,
                            SQLINTEGER FAR *pirow);
```

Function arguments

Table 115. SQLParamOptions arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLINTEGER	crow	Input	Number of values for each parameter. If this is greater than 1, then the <i>rgbValue</i> argument in SQLBindParameter() points to an array of parameter values, and <i>pcbValue</i> points to an array of lengths.
SQLINTEGER *	pirow	Output (deferred)	Pointer to the buffer for the current parameter array index. As each set of parameter values is processed, <i>pirow</i> is set to the array index of that set. If a statement fails, <i>pirow</i> can be used to determine how many statements were successfully processed. Nothing is returned if the <i>pirow</i> pointer is NULL.

Usage

DB2 ODBC prepares the statement, and executes it repeatedly for the array of parameter markers.

As a statement executes, *pirow* is set to the index of the current array of parameter values. If an error occurs during execution for a particular element in the array, execution halts and SQLExecute(), SQLExecDirect() or SQLParamData() returns SQL_ERROR.

The contents of *pirow* have the following uses:

- When SQLParamData() returns SQL_NEED_DATA, the application can access the value in *pirow* to determine which set of parameters is being assigned values.
- When SQLExecute() or SQLExecDirect() returns an error, the application can access the value in *pirow* to find out which element in the parameter value array failed.
- When SQLExecute(), SQLExecDirect(), SQLParamData(), or SQLPutData() succeeds, the value in *pirow* is set to the input value in *crow* to indicate that all elements of the array have been processed successfully.

The output argument *pirow* indicates how many sets of parameters were successfully processed. If the statement processed is a query, *pirow* indicates the array index associated with the current result set returned by `SQLMoreResults()` and is incremented each time `SQLMoreResults()` is called.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 116. SQLParamOptions SQLSTATEs

SQLSTATE	Description	Explanation
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (<code>SQLParamData()</code> , <code>SQLPutData()</code>) operation.
HY107	Row value out of range.	The value in the argument <i>crow</i> is less than 1.

Restrictions

None.

Example

See “Array input example” on page 405.

References

- “SQLBindParameter - Binds a parameter marker to a buffer or LOB locator” on page 91
- “SQLMoreResults - Determine if there are more result sets” on page 286
- “SQLSetStmtAttr - Set options related to a statement” on page 360

SQLPrepare - Prepare a statement

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the DBMS to be prepared. The application can reference this prepared statement by passing the statement handle to other functions.

If the statement handle has been previously used with a query statement (or any function that returns a result set), SQLCloseCursor() must be called to close the cursor, before calling SQLPrepare().

Syntax

```
SQLRETURN SQLPrepare(
    SQLHSTMT hstmt,
    SQLCHAR FAR *szSqlStr,
    SQLINTEGER cbSqlStr);
```

Function arguments

Table 117. SQLPrepare arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle. There must not be an open cursor associated with <i>hstmt</i> .
SQLCHAR *	<i>szSqlStr</i>	input	SQL statement string.
SQLINTEGER	<i>cbSqlStr</i>	input	Length of contents of <i>szSqlStr</i> argument. This must be set to either the exact length of the SQL statement in <i>szSqlStr</i> , or to SQL_NTS if the statement text is null-terminated.

Usage

If the SQL statement text contains vendor escape clause sequences, DB2 ODBC first modifies the SQL statement text to the appropriate DB2 specific format before submitting it to the database for preparation. If the application does not generate SQL statements that contain vendor escape clause sequences (see “Using vendor escape clauses” on page 448); then the SQL_NOSCAN statement option 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 was 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 “?” 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() and SQLSetParam() are used to bind (associate) application values with each parameter marker and to indicate if any data conversion should be performed at the time the data is transferred.

All parameters must be bound before calling SQLExecute(). For more information see “SQLExecute - Execute a statement” on page 166.

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 OS/390 and z/OS supports, see Table 1 on page 10.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement must be defined on a separate statement handle under the same connection handle and same isolation level.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 118. SQLPrepare SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	<i>szSql/Str</i> contains an UPDATE or DELETE statement which did not contain a WHERE clause.
21S01	Insert value list does not match column list.	<i>szSql/Str</i> contains an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	<i>szSql/Str</i> contains a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
34000	Invalid cursor name.	<i>szSql/Str</i> contains a positioned DELETE or a positioned UPDATE and the cursor referenced by the statement being executed is not open.
37xxx ^a	Invalid SQL syntax.	<i>szSql/Str</i> contains one or more of the following: <ul style="list-style-type: none"> • A COMMIT • A ROLLBACK • An SQL statement that the connected database server cannot prepare • A statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belongs is rolled back due to deadlock or timeout.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.

SQLPrepare

Table 118. SQLPrepare SQLSTATEs (continued)

SQLSTATE	Description	Explanation
42xxx ^a	Syntax error or access rule violation	425xx indicates the authorization ID does not have permission to execute the SQL statement contained in <i>szSqlStr</i> . Other 42xxx SQLSTATEs indicate a variety of syntax or access problems with the statement.
42S01	Database object already exists.	<i>szSqlStr</i> contains a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already exists.
42S02	Database object does not exist.	<i>szSqlStr</i> contains an SQL statement that references a table name or a view name that does not exist.
42S11	Index already exists.	<i>szSqlStr</i> contains a CREATE INDEX statement and the specified index name already exists.
42S12	Index not found.	<i>szSqlStr</i> contains a DROP INDEX statement and the specified index name does not exist.
42S21	Column already exists.	<i>szSqlStr</i> contains an ALTER TABLE statement and the column specified in the ADD clause is not unique or identifies an existing column in the base table.
42S22	Column not found.	<i>szSqlStr</i> contains an SQL statement that references a column name that does not exist.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	<i>szSqlStr</i> is a null pointer.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The argument <i>cbSqlStr</i> is less than 1, but not equal to SQL_NTS.

Note:

^a xxx refers to any SQLSTATE with that class code. Example, **37xxx** refers to any SQLSTATE in the **37** class.

Not all DBMSs report all of the above diagnostic messages at prepare time. Therefore, an application must also be able to handle these conditions when calling SQLExecute().

Restrictions

None.

Example

```

/*****
/*      Prepare a query and execute a 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      */
    *****/

    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;

```

SQLPrepare

```

/*****
/* 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;

/*****
/* 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)

```

SQLPrepare

```
{
    (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 */
*****/

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 CLIP07.\n\n");

return RETCODE;
}
```

References

- “SQLBindParameter - Binds a parameter marker to a buffer or LOB locator” on page 91
- “SQLColAttribute - Get column attributes” on page 106
- “SQLDescribeCol - Describe column attributes” on page 137
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLExecute - Execute a statement” on page 166
- “SQLNumParams - Get number of parameters in a SQL statement” on page 292
- “SQLNumResultCols - Get number of result columns” on page 294
- “SQLSetParam - Binds a parameter marker to a buffer” on page 354

SQLPrimaryKeys - Get primary key columns of a table

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLPrimaryKeys() returns a list of column names that comprise the primary key for a table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

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

Function arguments

Table 119. SQLPrimaryKeys arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szCatalogName	input	Catalog qualifier of a 3 part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbCatalogName	input	Length of <i>szCatalogName</i> .
SQLCHAR *	szSchemaName	input	Schema qualifier of table name.
SQLSMALLINT	cbSchemaName	input	Length of <i>szSchemaName</i> .
SQLCHAR *	szTableName	input	Table name.
SQLSMALLINT	cbTableName	input	Length of <i>szTableName</i> .

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 120 on page 309, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME and ORDINAL_POSITION.

Since calls to SQLPrimaryKeys() in many cases map to a complex and, thus, expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and

SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 120. Columns returned by SQLPrimaryKeys

Column number/name	Data type	Description
1 TABLE_CAT	VARCHAR(128)	This is always null.
2 TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
3 TABLE_NAME	VARCHAR(128) NOT NULL	Name of the specified table.
4 COLUMN_NAME	VARCHAR(128) NOT NULL	Primary key column name.
5 KEY_SEQ	SMALLINT NOT NULL	Column sequence number in the primary key, starting with 1.
6 PK_NAME	VARCHAR(128)	Primary key identifier. NULL if not applicable to the data source.

Note: 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

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 121. SQLPrimaryKeys SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal SQL_NTS.
HYC00	Driver not capable.	DB2 ODBC does not support <i>catalog</i> as a qualifier for table name.

Restrictions

None.

SQLPrimaryKeys

Example

The following example uses SQLPrimaryKeys to locate a primary key for a table, and calls SQLColAttributes to find its data type.

```
/* ... */

#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;
}

/*
 * Since 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_DESC_TYPE,
                    rgbDesc,
                    20,
                    &pdbcDesc,
                    &pfDesc );
if( rc != SQL_SUCCESS )
{
    goto exit;
}

/*
 * Display the data type.
 */
fprintf( stdout, "Data type ==> %s\n", rgbDesc );

```

SQLPrimaryKeys

```
exit:
/*****
/* Clean up the environment...
*****/

SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);

SQLDisconnect( hdbc );

SQLFreeHandle(SQL_HANDLE_DBC, hdbc);

SQLFreeHandle(SQL_HANDLE_ENV, henv);

}
```

References

- “SQLForeignKeys - Get the list of foreign key columns” on page 181
- “SQLStatistics - Get index and statistics information for a base table” on page 374

SQLProcedureColumns - Get input/output parameter information for a procedure

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLProcedureColumns() returns a list of input and output parameters associated with a procedure. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN SQLProcedureColumns (
    SQLHSTMT          hstmt,
    SQLCHAR           FAR *szProcCatalog,
    SQLSMALLINT       cbProcCatalog,
    SQLCHAR           FAR *szProcSchema,
    SQLSMALLINT       cbProcSchema,
    SQLCHAR           FAR *szProcName,
    SQLSMALLINT       cbProcName,
    SQLCHAR           FAR *szColumnName,
    SQLSMALLINT       cbColumnName);
```

Function arguments

Table 122. SQLProcedureColumns arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szProcCatalog	input	Catalog qualifier of a 3 part procedure name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbProcCatalog	input	Length of <i>szProcCatalog</i> . This must be set to 0.
SQLCHAR *	szProcSchema	input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by schema name. For Version 4 and Version 5 of DB2 for OS/390, all the stored procedures are in one schema; the only acceptable value for the <i>szProcSchema</i> argument is a null pointer. For DB2 UDB, <i>szProcSchema</i> can contain a valid pattern value. For more information about valid search patterns, see "Querying system catalog information" on page 397.
SQLSMALLINT	cbProcSchema	input	Length of <i>szProcSchema</i> .
SQLCHAR *	szProcName	input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by procedure name.
SQLSMALLINT	cbProcName	input	Length of <i>szProcName</i> .
SQLCHAR *	szColumnName	input	Buffer that can contain a <i>pattern-value</i> 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 <i>szProcName</i> and/or <i>szProcSchema</i> .
SQLSMALLINT	cbColumnName	input	Length of <i>szColumnName</i> .

SQLProcedureColumns

Usage

If the stored procedure is at a DB2 for MVS/ESA Version 4 server or a DB2 for OS/390 Version 5 server, the name of the stored procedures must be registered in the server's SYSIBM.SYSPROCEDURES catalog table. If the stored procedure is at a DB2 for OS/390 Version 6 server or later, the name of the stored procedures must be registered in the server's SYSIBM.SYSROUTINES catalog table.

For versions of other DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set is returned.

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 123 lists the columns in the result set.

Since calls to SQLProcedureColumns() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Applications should be aware that columns beyond the last column might be defined in future releases. Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 123. Columns returned by SQLProcedureColumns

Column number/name	Data type	Description
1 PROCEDURE_CAT	VARCHAR(128)	The is always null.
2 PROCEDURE_SCHEM	VARCHAR(128)	The name of the schema containing PROCEDURE_NAME. (This is also NULL for DB2 for OS/390 and z/OS SQLProcedureColumns() result sets.)
3 PROCEDURE_NAME	VARCHAR(128)	Name of the procedure.
4 COLUMN_NAME	VARCHAR(128)	Name of the parameter.

Table 123. Columns returned by SQLProcedureColumns (continued)

Column number/name	Data type	Description
5 COLUMN_TYPE	SMALLINT NOT NULL	<p>Identifies the type information associated with this row. The values can be:</p> <ul style="list-style-type: none"> SQL_PARAM_TYPE_UNKNOWN: the parameter type is unknown. Note: This is not returned. SQL_PARAM_INPUT: this parameter is an input parameter. SQL_PARAM_INPUT_OUTPUT: this parameter is an input / output parameter. SQL_PARAM_OUTPUT: this parameter is an output parameter. SQL_RETURN_VALUE: the procedure column is the return value of the procedure. Note: This is not returned. SQL_RESULT_COL: this parameter is actually a column in the result set. Note: This is not returned. Note: SQL_PARAM_OUTPUT and SQL_RETURN_VALUE are supported only on ODBC 2.0 or higher.
6 DATA_TYPE	SMALLINT NOT NULL	SQL data type.
7 TYPE_NAME	VARCHAR(128) NOT NULL	Character string representing the name of the data type corresponding to DATA_TYPE.
8 COLUMN_SIZE	INTEGER	<p>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.</p> <p>For date, time, timestamp data types, this is the total number of bytes required to display the value when converted to character.</p> <p>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.</p> <p>See Table 174 on page 486.</p>
9 BUFFER_LENGTH	INTEGER	<p>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 null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.</p> <p>See Table 176 on page 488.</p>
10 DECIMAL_DIGITS	SMALLINT	<p>The scale of the parameter. NULL is returned for data types where scale is not applicable.</p> <p>See Table 175 on page 487.</p>

SQLProcedureColumns

Table 123. Columns returned by SQLProcedureColumns (continued)

Column number/name	Data type	Description
11 NUM_PREC_RADIX	SMALLINT	<p>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.</p> <p>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.</p> <p>For numeric data types, the DBMS can return a NUM_PREC_RADIX of either 10 or 2.</p> <p>NULL is returned for data types where radix is not applicable.</p>
12 NULLABLE	SMALLINT NOT NULL	<p>SQL_NO_NULLS if the parameter does not accept NULL values.</p> <p>SQL_NULLABLE if the parameter accepts NULL values.</p>
13 REMARKS	VARCHAR(254)	Might contain descriptive information about the parameter.
14 COLUMN_DEF	VARCHAR(254)	<p>The column's default value. If the default value is:</p> <ul style="list-style-type: none"> A numeric literal, this column contains the character representation of the numeric literal with no enclosing single quotes. A character string, this column is that string enclosed in single quotes. A pseudo-literal, such as for DATE, TIME, and TIMESTAMP columns, this column contains the keyword of the pseudo-literal (for example, CURRENT DATE) with no enclosing single quotes. NULL, this column returns the word NULL, with no enclosing single quotes. <p>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.</p>
15 SQL_DATA_TYPE	SMALLINT NOT NULL	The SQL data type. This column is the same as the DATA_TYPE column. For datetime data types, the SQL_DATA_TYPE field in the result set is SQL_DATETIME, and the SQL_DATETIME_SUB field returns the subcode for the specific datetime data type (SQL_CODE_DATE, SQL_CODE_TIME or SQL_CODE_TIMESTAMP).
16 SQL_DATETIME_SUB	SMALLINT	<p>The subtype code for datetime data types:</p> <ul style="list-style-type: none"> SQL_CODE_DATE SQL_CODE_TIME SQL_CODE_TIMESTAMP <p>For all other data types, this column returns a NULL.</p>
17 CHAR_OCTET_LENGTH	INTEGER	The maximum length in bytes of a character data type column. For all other data types, this column returns a NULL.
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 123. Columns returned by SQLProcedureColumns (continued)

Column number/name	Data type	Description
19 IS_NULLABLE	VARCHAR(128)	<p>One of the following:</p> <ul style="list-style-type: none"> • "NO", if the column does not include NULLs • "YES", if the column can include NULLs • Zero-length string if nullability is unknown. <p>The value returned for this column is different than the value returned for the NULLABLE column. (See the description of the NULLABLE column.)</p>

Note: 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

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 124. SQLProcedureColumns SQLSTATES

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
42601	PARMLIST syntax error.	The PARMLIST value in the stored procedures catalog table contains a syntax error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length	The value of one of the name length arguments is less than 0, but not equal SQL_NTS.
HYC00	Driver not capable.	<p>DB2 ODBC does not support <i>catalog</i> as a qualifier for procedure name.</p> <p>The connected server does not support <i>schema</i> as a qualifier for procedure name.</p>

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.

SQLProcedureColumns

Example

```
/* *****  
/* 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);

```

SQLProcedureColumns

```
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;

```

SQLProcedureColumns

```

/*****
/* 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;
}

```

References

- “SQLProcedures - Get list of procedure names” on page 323

SQLProcedures - Get list of procedure names

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLProcedures () returns a list of procedure names that have been registered at the server, and which match the specified search pattern.

The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN SQLProcedures (SQLHSTMT hstmt,
                          SQLCHAR FAR *szProcCatalog,
                          SQLSMALLINT cbProcCatalog,
                          SQLCHAR FAR *szProcSchema,
                          SQLSMALLINT cbProcSchema,
                          SQLCHAR FAR *szProcName,
                          SQLSMALLINT cbProcName);
```

Function arguments

Table 125. SQLProcedures arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLCHAR *	szProcCatalog	Input	Catalog qualifier of a 3 part procedure name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbProcCatalog	Input	Length of <i>szProcCatalog</i> . This must be set to 0.
SQLCHAR *	szProcSchema	Input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by schema name. For Version 4 and Version 5 of DB2 for OS/390, all the stored procedures are in one schema; the only acceptable value for the <i>szProcSchema</i> argument is a null pointer. For DB2 UDB, <i>szProcSchema</i> can contain a valid pattern value. For more information about valid search patterns, see "Querying system catalog information" on page 397.
SQLSMALLINT	cbProcSchema	Input	Length of <i>szProcSchema</i> .
SQLCHAR *	szProcName	Input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by table name.
SQLSMALLINT	cbProcName	Input	Length of <i>szProcName</i> .

Usage

If the stored procedure is at a DB2 for MVS/ESA Version 4 server or a DB2 for OS/390 Version 5 server, the name of the stored procedures must be registered in the servers SYSIBM.SYSPROCEDURES catalog table. If the stored procedure is at a DB2 for OS/390 Version 6 server or later, the name of the stored procedure must be registered in the servers SYSIBM.SYSROUTINES catalog table.

SQLProcedures

For other versions of DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set is returned.

The result set returned by `SQLProcedures()` contains the columns listed in Table 126 in the order given. The rows are ordered by `PROCEDURE_CAT`, `PROCEDURE_SCHEMA`, and `PROCEDURE_NAME`.

Since calls to `SQLProcedures()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The `VARCHAR` columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the `SQL_MAX_CATALOG_NAME_LEN`, `SQL_MAX_SCHEMA_NAME_LEN`, `SQL_MAX_TABLE_NAME_LEN`, and `SQL_MAX_COLUMN_NAME_LEN` to determine respectively the actual lengths of the `TABLE_CAT`, `TABLE_SCHEM`, `TABLE_NAME`, and `COLUMN_NAME` columns supported by the connected DBMS.

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 126. Columns returned by `SQLProcedures`

Column number/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: <ul style="list-style-type: none">• <code>SQL_PT_UNKNOWN</code>: It cannot be determined whether the procedure returns a value.• <code>SQL_PT_PROCEDURE</code>: The returned object is a procedure; that is, it does not have a return value.• <code>SQL_PT_FUNCTION</code>: The returned object is a function; that is, it has a return value. DB2 ODBC always returns <code>SQL_PT_PROCEDURE</code> .

Note: 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

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`

Diagnostics

Table 127. SQLProcedures SQLSTATES

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal to SQL_NTS.
HYC00	Driver not capable.	DB2 ODBC does not support <i>catalog</i> as a qualifier for procedure name. The connected server does not supported schema as a qualifier for procedure name.

Restrictions

If an application is connected to a DB2 server that does not provide support for stored procedures, or for a stored procedure catalog, `SQLProcedureColumns()` returns an empty result set.

Example

```

/* ... */

printf("Enter Procedure Schema Name Search Pattern:\n");
gets(proc_schem.s);

rc = SQLProcedures(hstmt, NULL, 0, proc_schem.s, SQL_NTS, "%", SQL_NTS);

rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) proc_schem.s, 129,
                &proc_schem.ind);

rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) proc_name.s, 129,
                &proc_name.ind);

rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) remarks.s, 255,
                &remarks.ind);

printf("PROCEDURE SCHEMA          PROCEDURE NAME          \n");
printf("----- \n");
/* Fetch each row, and display */
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
    printf("%-25s %-25s\n", proc_schem.s, proc_name.s);
    if (remarks.ind != SQL_NULL_DATA) {
        printf(" (Remarks) %s\n", remarks.s);
    }
}
/* endwhile */
/* ... */

```

SQLProcedures

References

- “SQLProcedureColumns - Get input/output parameter information for a procedure” on page 313

SQLPutData - Passing data value for a parameter

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

SQLPutData() is called following an SQLParamData() call returning SQL_NEED_DATA to supply parameter data values. This function can be used to send large parameter values in pieces.

The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN SQLPutData(
    (SQLHSTMT
     SQLPOINTER
     SQLINTEGER)
    hstmt,
    rgbValue,
    cbValue);
```

Function arguments

Table 128. SQLPutData arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLPOINTER	rgbValue	Input	Pointer to the actual data, or portion of data, for a parameter. The data must be in the form specified in the SQLBindParameter() call that the application used when specifying the parameter.
SQLINTEGER	cbValue	Input	Length of <i>rgbValue</i> . Specifies the amount of data sent in a call to SQLPutData() . The amount of data can vary with each call for a given parameter. The application can also specify SQL_NTS or SQL_NULL_DATA for <i>cbValue</i> . <i>cbValue</i> is ignored for all fixed length C buffer types, such as date, time, timestamp, and all numeric C buffer types. For cases where the C buffer type is SQL_C_CHAR or SQL_C_BINARY, or if SQL_C_DEFAULT is specified as the C buffer type and the C buffer type default is SQL_C_CHAR or SQL_C_BINARY, this is the number of bytes of data in the <i>rgbValue</i> buffer.

Usage

For a description on the SQLParamData() and SQLPutData() sequence, see "Sending/retrieving long data in pieces" on page 401.

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

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()

Should they be invoked during an SQL_NEED_DATA sequence, these functions return SQL_ERROR with SQLSTATE of HY010 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 22005. This error does not result in a change of state; the statement handle is still in a *Need Data* state and the application can continue sending parameter data.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_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.

Table 129. SQLPutData SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data sent for a numeric parameter is truncated without the loss of significant digits. Timestamp data sent for a date or time column is truncated. Function returns with SQL_SUCCESS_WITH_INFO.
22001	String data right truncation.	More data is sent for a binary or char data than the data source can support for that column.
22008	Invalid datetime format or datetime field overflow.	The data value sent for a date, time, or timestamp parameters is invalid.
22018	Error in assignment.	The data sent for a parameter is incompatible with the data type of the associated table column.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	The argument <i>rgbValue</i> is a NULL pointer, and the argument <i>cbValue</i> is neither 0 nor SQL_NULL_DATA.

Table 129. SQLPutData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The statement handle <i>hstmt</i> must be in a need data state and must have been positioned on an SQL_DATA_AT_EXEC parameter using a previous SQLParamData() call.
HY019	Numeric value out of range.	The data sent for a numeric parameter cause the whole part of the number to be truncated when assigned to the associated column. SQLPutData() was called more than once for a fixed length parameter.
HY090	Invalid string or buffer length.	The argument <i>rgbValue</i> is not a NULL pointer, and the argument <i>cbValue</i> 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. Since 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 of the entire data that would be sent for character or binary C data using the subsequent SQLPutData() calls. Since the DB2 ODBC driver does not need this information, the macro is not needed. An ODBC application calls SQLGetInfo() with the SQL_NEED_LONG_DATA_LEN option to check if the driver needs this information. The DB2 ODBC driver returns 'N' to indicate that this information is not needed by SQLPutData().

Example

See “Example” on page 214.

References

- “SQLBindParameter - Binds a parameter marker to a buffer or LOB locator” on page 91
- “SQLExecute - Execute a statement” on page 166
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLParamData - Get next parameter for which a data value is needed” on page 296
- “SQLCancel - Cancel statement” on page 102

SQLRowCount - Get row count

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLRowCount() returns the number of rows in a table that were affected by an UPDATE, INSERT, or DELETE statement executed against the table, or a view based on the table.

SQLExecute() or SQLExecDirect() must be called before calling this function.

Syntax

```
SQLRETURN SQLRowCount (SQLHSTMT hstmt,
                        SQLINTEGER FAR *pcrow);
```

Function arguments

Table 130. SQLRowCount arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle
SQLINTEGER *	<i>pcrow</i>	output	Pointer to location where the number of rows affected is stored.

Usage

If the last executed statement referenced by the input statement handle was not an UPDATE, INSERT, or DELETE statement, or if it did not execute successfully, then the function sets the contents of *pcrow* to -1.

If SQLRowCount() is executed after the SQLExecDirect() or SQLExecute() of an SQL statement other than INSERT, UPDATE, or DELETE, it results in return code 0 and *pcrow* is set to -1.

Any rows in other tables that might be affected by the statement (for example, cascading deletes) are not included in the count.

If SQLRowCount() is executed after a built-in function (for example, SQLTables()), it results in return code -1 and SQLSTATE S1010.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 131. SQLRowCount SQLSTATES

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

Table 131. SQLRowCount SQLSTATEs (continued)

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called prior to calling SQLExecute() or SQLExecDirect() for the <i>hstmt</i> .
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

See “Example” on page 139.

References

- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLExecute - Execute a statement” on page 166
- “SQLNumResultCols - Get number of result columns” on page 294

SQLSetColAttributes - Set column attributes

Purpose

Specification:			
-----------------------	--	--	--

SQLSetColAttributes() sets the data source result descriptor (column name, type, precision, scale and nullability) for one column in the result set so that the DB2 ODBC implementation does not have to obtain the descriptor information from the DBMS server.

Syntax

```
SQLRETURN SQLSetColAttributes (SQLHSTMT      hstmt,
                               SQLUSMALLINT   icol,
                               SQLCHAR        FAR *pszColName,
                               SQLSMALLINT     cbColName,
                               SQLSMALLINT     fSQLType,
                               SQLINTEGER      cbColDef,
                               SQLSMALLINT     ibScale,
                               SQLSMALLINT     fNullable);
```

Function arguments

Table 132. 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	Length of szColName buffer.

Table 132. SQLSetColAttributes arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	fSqlType	input	The SQL data type of the column. The following values are recognized: <ul style="list-style-type: none"> • SQL_BINARY • SQL_BLOB • SQL_CHAR • SQL_CLOB • SQL_DBCLOB • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_ROWID • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC
SQLINTEGER	cbColDef	input	The precision of the column on the data source.
SQLSMALLINT	ibScale	input	The scale of the column on the data source. This is ignored for all data types except SQL_DECIMAL, SQL_NUMERIC, SQL_TYPE_TIMESTAMP.
SQLSMALLINT	fNullable	input	Indicates whether the column allows NULL value. This must be one of the following values: <ul style="list-style-type: none"> • SQL_NO_NULLS - the column does not allow NULL values. • SQL_NULLABLE - the column allows NULL values.

Usage

This function is designed to help reduce the amount of network traffic that can result when an application is fetching result sets that contain an extremely large number of columns. If the application has advanced knowledge of the characteristics of the descriptor information of a result set (that is, the exact number of columns, column name, data type, nullability, precision, or scale), then it can inform DB2 ODBC rather than having DB2 ODBC obtain this information from the database, thus reducing the quantity of network traffic.

An application typically calls `SQLSetColAttributes()` after a call to `SQLPrepare()` and before the associated call to `SQLExecute()`. An application can also call `SQLSetColAttributes()` before a call to `SQLExecDirect()`. This function is valid only after the statement option `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.

SQLSetColAttributes

Recommendation: The application should only use this function if it has prior knowledge of 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 07002). 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

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 133. SQLSetColAttributes SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	szColName contains a column name that is too long. To obtain the maximum length of the column name, call SQLGetInfo with the flnfoType SQL_MAX_COLUMN_NAME_LEN.
24000	Invalid cursor state.	A cursor is already open on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY000	General error.	An error occurred for which there is no specific SQLSTATE and for which no implementation defined SQLSTATE is defined. The error message returned by SQLGetDiagRec() in the argument szErrorMsg describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY004	Invalid SQL data type.	The value specified for the argument fSqlType is not a valid SQL data type.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument cbColName is less than 0 and not equal to SQL_NTS.
HY099	Nullable type out of range.	The value specified for fNullable is invalid.
HY104	Invalid precision or scale value.	The value specified for fSqlType is either SQL_DECIMAL or SQL_NUMERIC and the value specified for cbColDef is less than 1, or the value specified for ibScale is less than 0 or greater than the value for the argument cbColDef (precision). The value specified for fSqlType is SQL_TYPE_TIMESTAMP and the value for ibScale is less than 0 or greater than 6.
S1002	Invalid column number.	The value specified for the argument icol is less than 1 or greater than the maximum number of columns supported by the server.

Restrictions

None.

Example

```

/* ... */
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 */

```

References

- “SQLColAttribute - Get column attributes” on page 106
- “SQLDescribeCol - Describe column attributes” on page 137
- “SQLExecute - Execute a statement” on page 166
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLPrepare - Prepare a statement” on page 300

SQLSetConnectAttr - Set connection attributes

Purpose

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLSetConnectAttr() sets attributes that govern aspects of connections.

Syntax

```
SQLRETURN SQLSetConnectAttr (SQLHDBC          ConnectionHandle,
                             SQLINTEGER       Attribute,
                             SQLPOINTER      ValuePtr,
                             SQLINTEGER      StringLength);
```

Function arguments

Table 134. SQLSetConnectAttr arguments

Data type	Argument	Use	Description
SQLHDBC	<i>ConnectionHandle</i>	input	Connection handle.
SQLINTEGER	<i>Attribute</i>	input	Connection attribute to set. Refer to Table 136 on page 337 for a complete list of attributes.
SQLPOINTER	<i>ValuePtr</i>	input	Pointer to the value to be associated with <i>Attribute</i> . Depending on the value of <i>Attribute</i> , <i>*ValuePtr</i> will be a 32-bit unsigned integer value or point to a null-terminated character string. If the <i>Attribute</i> argument is a driver-specific value, the value in <i>*ValuePtr</i> might be a signed integer.
SQLINTEGER	<i>StringLength</i>	input	Information about the <i>*ValuePtr</i> argument. <ul style="list-style-type: none"> • For ODBC-defined attributes: <ul style="list-style-type: none"> – If <i>ValuePtr</i> points to a character string, this argument should be the length of <i>*ValuePtr</i>. – If <i>ValuePtr</i> points to an integer, <i>BufferLength</i> is ignored. • For driver-defined attributes (IBM extension): <ul style="list-style-type: none"> – If <i>ValuePtr</i> points to a character string, this argument should be the length of <i>*ValuePtr</i> or SQL_NTS if it is a null-terminated string. – If <i>ValuePtr</i> points to an integer, <i>BufferLength</i> is ignored.

Usage

SQLSetConnectAttr() sets attributes that govern aspects of connections.

An application can call SQLSetConnectAttr() at any time between the time the connection is allocated or freed. All connection and statement attributes successfully set by the application for the connection persist until SQLFreeHandle() is called on the connection.

Some connection attributes can be set only before or after a connection is made. Other attributes cannot be set after a statement is allocated. Table 135 on page 337 indicates when each of the connection attributes can be set.

Table 135. When connection attributes can be set

<i>Attribute</i>	Before connection	After connection	After statements allocated
SQL_ATTR_ACCESS_MODE	Yes	Yes	Yes ¹
SQL_ATTR_AUTOCOMMIT	Yes	Yes	Yes ²
SQL_ATTR_CONNECTTYPE	Yes	No	No
SQL_ATTR_CURRENT_SCHEMA	Yes	Yes	Yes
SQL_ATTR_MAXCONN	Yes	No	No
SQL_ATTR_SYNC_POINT	Yes	No	No
SQL_ATTR_TXN_ISOLATION	No	Yes	Yes

Notes:

1. Attribute only affects subsequently allocated statements.
2. Attribute can be set only if there are no open transactions on the connections.

Table 136 lists the SQLSetConnectAttr() *Attribute* values. DB2 ODBC supports all of the ODBC 2.0 *Attribute* values that are renamed in ODBC 3.0.

For a summary of the *Attribute* values renamed in ODBC 3.0, see Table 196 on page 502.

ODBC applications that need to set statement options should use SQLSetStmtAttr(). Although the ability to set statement options on the connect level is supported, it is not recommended.

Table 136. Connect options

<i>Attribute</i>	<i>ValuePtr</i>
SQL_ATTR_ACCESS_MODE	<p>A 32-bit integer value which can be either:</p> <ul style="list-style-type: none"> • SQL_MODE_READ_ONLY: Indicates that the application is not performing any updates on data from this point on. Therefore, a less restrictive isolation level and locking can be used on transactions; that is, uncommitted read (SQL_TXN_READ_UNCOMMITTED). <p>DB2 ODBC does not ensure that requests to the database are <i>read-only</i>. 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.</p> <ul style="list-style-type: none"> • SQL_MODE_READ_WRITE: Indicates that the application is making updates on data from this point on. DB2 ODBC goes back to using the default transaction isolation level for this connection. <p>SQL_MODE_READ_WRITE is the default.</p>

There must not be any outstanding transactions on this connection.

SQLSetConnectAttr

Table 136. Connect options (continued)

Attribute	ValuePtr
SQL_ATTR_AUTOCOMMIT	<p data-bbox="613 258 1409 310">A 32-bit integer value that specifies whether to use auto-commit or manual commit mode:</p> <ul data-bbox="613 323 1409 499" style="list-style-type: none"><li data-bbox="613 323 1409 375">• SQL_AUTOCOMMIT_OFF: The application must manually, explicitly commit or rollback transactions with <code>SQLEndTran()</code> calls.<li data-bbox="613 388 1409 499">• SQL_AUTOCOMMIT_ON: DB2 ODBC operates in auto-commit mode. Each statement is implicitly committed. Each statement, that is not a query, is committed immediately after it has been executed. Each query is committed immediately after the associated cursor is closed. <p data-bbox="639 512 1062 533">SQL_AUTOCOMMIT_ON is the default.</p> <p data-bbox="639 541 1425 594">Note: If this is a coordinated distributed unit of work connection, then the default is SQL_AUTOCOMMIT_OFF</p> <p data-bbox="639 606 1393 720">When specifying auto-commit, the application can have only one outstanding statement per connection. For example, there must not be two open cursors, or unpredictable results can occur. An open cursor must be closed before another query is executed.</p> <p data-bbox="613 747 1425 858">Since 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 auto-commit mode.</p> <p data-bbox="613 888 1377 940">Changing from manual-commit to auto-commit mode commits any open transaction on the connection.</p>

Table 136. Connect options (continued)

Attribute	ValuePtr
SQL_ATTR_CONNECTTYPE	<p data-bbox="646 254 1458 369">A 32-bit integer value that specifies whether this application is to operate in a coordinated or uncoordinated distributed environment. If the processing needs to be coordinated, then this option must be considered in conjunction with the SQL_SYNC_POINT connection option. The possible values are:</p> <ul data-bbox="646 380 1458 869" style="list-style-type: none"> <li data-bbox="646 380 1458 558">• SQL_CONCURRENT_TRANS: The application can have concurrent multiple connections to any one database or to multiple databases. This option setting corresponds to the specification of the type 1 CONNECT in embedded SQL. Each connection has its own commit scope. No effort is made to enforce coordination of transaction. The current setting of the SQL_SYNC_POINT option is ignored. This is the default. <li data-bbox="646 611 1458 869">• SQL_COORDINATED_TRANS: The application wishes to have commit and rollbacks coordinated among multiple database connections. This option setting corresponds to the specification of the type 2 CONNECT in embedded SQL and must be considered in conjunction with the SQL_SYNC_POINT connection option. In contrast to the SQL_CONCURRENT_TRANS setting described above, the application is permitted only one open connection per database. Note: This connection type results in the default for SQL_AUTOCOMMIT connection option to be SQL_AUTOCOMMIT_OFF. <p data-bbox="646 894 1458 953">This option must be set before making a connect request; otherwise, the SQLSetConnectAttr() call is rejected.</p> <p data-bbox="646 978 1458 1089">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.</p> <p data-bbox="646 1115 1458 1262">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.</p> <p data-bbox="646 1287 1073 1310">Note: This is an IBM-defined extension.</p>
SQL_ATTR_CURRENT_SCHEMA	<p data-bbox="646 1325 1458 1415">A null-terminated character string containing the name of the schema to be used by DB2 ODBC for the SQLColumns() call if the <i>szSchemaName</i> pointer is set to null.</p> <p data-bbox="646 1440 1458 1499">To reset this option, specify this option with a zero length or a null pointer for the <i>vParam</i> argument.</p> <p data-bbox="646 1524 1458 1614">This option is useful when the application developer has coded a generic call to SQLColumns() that does not restrict the result set by schema name, but needs to constrain the result set at isolated places in the code.</p> <p data-bbox="646 1640 1458 1698">This option can be set at any time and is effective on the next SQLColumns() call where the <i>szSchemaName</i> pointer is null.</p> <p data-bbox="646 1724 1073 1747">Note: This is an IBM-defined extension.</p>

SQLSetConnectAttr

Table 136. Connect options (continued)

Attribute	ValuePtr
SQL_ATTR_MAXCONN	<p>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.</p> <p>This can be used as a governor for the maximum number of connections on a per application basis.</p> <p>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.</p> <p>Recommendation: Have the application set SQL_ATTR_MAXCONN at the environment level rather than on a connection basis. ODBC applications must set this attribute at the connection level since SQLSetEnvAttr() is not supported in ODBC.</p> <p>Note: This is an IBM-defined extension.</p>
SQL_ATTR_PARAMOPT_ATOMIC	<p>If specified, DB2 ODBC returns S1C00 on SQLSetConnectAttr() and HY011 on SQLGetConnectAttr().</p>
SQL_ATTR_SYNC_POINT	<p>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:</p> <ul style="list-style-type: none">• SQL_ONEPHASE: The DB2 ODBC 3.0 driver does not support SQL_ONEPHASE.• SQL_TWOPHASE: Two-phase commit is used to commit the work done by each database in a multiple database transaction. This requires the use of a transaction manager to coordinate two-phase commits among the databases that support this protocol. Multiple readers and multiple updaters are allowed within a transaction. This attribute is only utilized when SQL_ATTR_CONNECTTYPE attribute is SQL_ATTR_COORDINATED_TRANS. Then SQL_TWOPHASE is the default. This attribute is ignored when SQL_ATTR_CONNECTTYPE is set to SQL_ATTR_CONCURRENT_TRANS. See <i>DB2 SQL Reference</i> for more information about distributed unit of work transactions. <p>This option must be set before a connect request. Otherwise the option set request is rejected.</p> <p>All the connections within an application must have the same SQL_ATTR_CONNECTTYPE and SQL_ATTR_SYNC_POINT values. The first connection determines the acceptable attributes for the subsequent connections. Recommendation: The application should set the SQL_ATTR_CONNECTTYPE attribute at the environment level rather than at a connection level.</p>

Table 136. Connect options (continued)

Attribute	ValuePtr
SQL_ATTR_TXN_ISOLATION	<p>A 32-bit bitmask that sets the transaction isolation level for the current connection referenced by <i>hdbc</i>. The valid values for <i>vParam</i> can be determined at runtime by calling <code>SQLGetInfo()</code> with <i>fInfoType</i> set to <code>SQL_TXN_ISOLATION_OPTIONS</code>. The following values are accepted by DB2 ODBC, but each server might only support a subset of these isolation levels:</p> <ul style="list-style-type: none"> • <code>SQL_TXN_READ_UNCOMMITTED</code> - Dirty reads, reads that cannot be repeated, and phantoms are possible. • <code>SQL_TXN_READ_COMMITTED</code> - Dirty reads are not possible. Reads that cannot be repeated, and phantoms are possible. This is the default. • <code>SQL_TXN_REPEATABLE_READ</code> - Dirty reads and reads that cannot be repeated are not possible. Phantoms are possible. • <code>SQL_TXN_SERIALIZABLE</code> - Transactions can be serialized. Dirty reads, non-repeatable reads, and phantoms are not possible. • <code>SQL_TXN_NOCOMMIT</code> - Any changes are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed. This is analogous to autocommit. This is not an SQL92 isolation level, but an IBM defined extension, supported only by DB2 for AS/400. <p>In IBM terminology,</p> <ul style="list-style-type: none"> • <code>SQL_TXN_READ_UNCOMMITTED</code> is uncommitted read; • <code>SQL_TXN_READ_COMMITTED</code> is cursor stability; • <code>SQL_TXN_REPEATABLE_READ</code> is read stability; • <code>SQL_TXN_SERIALIZABLE</code> is repeatable read. <p>For a detailed explanation of isolation levels, see <i>IBM SQL Reference</i>.</p> <p>This option cannot be specified while there is an open cursor on any <code>hstmt</code>, or an outstanding transaction for this connection; otherwise, <code>SQL_ERROR</code> is returned on the function call (<code>SQLSTATE S1011</code>).</p> <p>Note: There is an IBM extension that permits the setting of transaction isolation levels on a per statement handle basis. See the <code>SQL_STMTTXN_ISOLATION</code> option in the function description for <code>SQLSetStmtAttr()</code>.</p>

Return codes

- `SQL_SUCCESS`
- `SQL_INVALID_HANDLE`
- `SQL_ERROR`

Diagnostics

Table 137. SQLSetConnectAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns <code>SQL_SUCCESS_WITH_INFO</code> .)
01S02	Option value changed.	<code>SQL_ATTR_SYNC_POINT</code> changed to <code>SQL_TWOPHASE</code> . <code>SQL_ONEPHASE</code> is not supported.
08S01	Unable to connect to data source.	The communication link between the application and the data source failed before the function completed.
08003	Connection is closed.	An <i>Attribute</i> value was specified that required an open connection, but the <i>ConnectionHandle</i> was not in a connected state.

SQLSetConnectAttr

Table 137. SQLSetConnectAttr SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 ODBC was not able to allocate memory for the specified handle.
HY009	Invalid use of a null pointer.	A null pointer was passed for <i>ValuePtr</i> and the value in <i>*ValuePtr</i> was a string value.
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() was called with the statement handle, and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY011	Operation invalid at this time.	The argument <i>Attribute</i> was SQL_ATTR_TXN_ISOLATION and a transaction was open.
HY024	Invalid attribute value.	Given the specified <i>Attribute</i> value, an invalid value was specified in <i>*ValuePtr</i> .
HY090	Invalid string or buffer length.	The StringLength argument was less than 0, but was not SQL_NTS.
HY092	Option type out of range.	The value specified for the argument <i>Attribute</i> was not valid for this version of DB2 ODBC.
HYC00	Driver not capable.	The value specified for the argument <i>Attribute</i> was a valid connection or statement attribute for this version of the DB2 ODBC driver, but was not supported by the data source.

Restrictions

None.

Example

```
rc=SQLSetConnectAttr( hdbc,SQL_ATTR_AUTOCOMMIT,  
                    (void*) SQL_AUTOCOMMIT_OFF, SQL_NTS);  
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
```

References

- “SQLGetConnectAttr - Get current attribute setting” on page 199
- “SQLSetStmtAttr - Set options related to a statement” on page 360
- “SQLAllocHandle - Allocate handle” on page 79

SQLSetConnection - Set connection handle

Purpose

Specification:			
-----------------------	--	--	--

This function is needed if the application needs to deterministically switch to a particular connection before continuing execution. It should only be used when the application is mixing DB2 ODBC function calls with embedded SQL function calls and multiple connections are involved.

Syntax

```
SQLRETURN SQLSetConnection (SQLHDBC          hdbc);
```

Function arguments

Table 138. SQLSetConnection arguments

Data type	Argument	Use	Description
SQLHDBC	hdbc	input	The connection handle associated with the connection to which the application wishes to switch.

Usage

ODBC allows multiple concurrent connections. It is not clear which connection an embedded SQL routine uses when invoked. In practice, the embedded routine uses the connection associated with the most recent network activity. However, from the application's perspective, this is not always easy to determine and it is difficult to keep track of this information. SQLSetConnection() is used to allow the application to *explicitly* specify which connection is active. The application can then call the embedded SQL routine.

SQLSetConnection() is not needed at all if the application makes purely DB2 ODBC calls. This is because each statement handle is implicitly associated with a connection handle and there is never any confusion as to which connection a particular DB2 ODBC function applies.

For more information on using embedded SQL within DB2 ODBC applications see "Mixing embedded SQL and DB2 ODBC" on page 446.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 139. SQLSetConnection SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection is closed.	The connection handle provided is not currently associated with an open connection to a database server.

SQLSetConnection

Table 139. SQLSetConnection SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY000	General error.	An error occurred for which there is no specific SQLSTATE and for which the implementation does not define an SQLSTATE. SQLGetDiagRec() returns an error message in the argument <i>szErrorMsg</i> that describes the error and its cause.

Restrictions

None.

Example

See “Mixed embedded SQL and DB2 ODBC example” on page 447.

References

- “SQLConnect - Connect to a data source” on page 129
- “SQLDriverConnect - (Expanded) connect to a data source” on page 146

SQLSetConnectOption - Set connection option

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	
-----------------------	-----------------	-------------------	--

In ODBC 3.0, SQLSetConnectAttr() replaces the ODBC 2.0 function SQLSetConnectOption(). See SQLSetConnectAttr() for more information.

SQLSetConnectOption() sets connection attributes for a particular connection.

Syntax

```
SQLRETURN SQLSetConnectOption(
    SQLHDBC          hdbc,
    SQLUSMALLINT     fOption,
    SQLINTEGER        vParam);
```

Function arguments

Table 140. SQLSetConnectOption arguments

Data Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
SQLUSMALLINT	fOption	Input	Connect option to set.
SQLINTEGER	vParam	Input	Value associated with <i>fOption</i> . Depending on the option, this can be a 32-bit integer value, or a pointer to a null-terminated string.

Usage

The SQLSetConnectOption() can be used to specify statement options for *all* statement handles on this connection, as well as for all future statement handles on this connection. For a list of statement options, see Table 150 on page 361.

All connection and statement options set using the SQLSetConnectOption() persist until SQLFreeConnect() is called or the next SQLSetConnectOption() call.

It is illegal to call SQLSetConnectOption() (SQLSTATE S1010) if any of the statement handles associated with this connection is in a need data state (that is, in the middle of an SQLParamData() -- SQLPutData() sequence to process SQL_DATA_AT_EXEC parameters). This sequence is described in "Sending/retrieving long data in pieces" on page 401.

The format of information set with *vParam* depends on the specified *fOption*. The option information can be either a 32-bit integer or a pointer to a null-terminated character string. In the case of the null-terminated character string, the maximum length of the string can be SQL_MAX_OPTION_STRING_LENGTH bytes (excluding the null-terminator).

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

SQLSetConnectOption

Diagnostics

Table 141. SQLSetConnectOption SQLSTATES

SQLSTATE	Description	Explanation
Note: Since SQLSetConnectOption() can also be used to set statement options, SQLSTATES for SQLSetConnectOption() can also include those listed under "Diagnostics" for the SQLSetStmtOption() API.		
01000	Warning.	Informational message indicating an internal commit has been issued on behalf of the application as part of the processing to set the specified connection option.
01S02	Option value changed.	SQL_CONNECTTYPE changed to SQL_CONCURRENT_TRANS when MULTICONTEXT=1 in use.
08003	Connection is closed.	An <i>fOption</i> is specified that required an open connection.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	Given the <i>fOption</i> value, an invalid value is specified for the argument <i>vParam</i> .
S1010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
S1011	Operation invalid at this time.	The <i>fOption</i> specified option cannot be set at this time: <ul style="list-style-type: none">• SQL_CONNECTTYPE: attempt is made to change the value of this option from its current value but the connection is open.• SQL_TXN_ISOLATION, SQL_ACCESS_MODE: a transaction is outstanding.
S1092	Option type out of range.	An invalid <i>fOption</i> value is specified.
S1C00	Driver not capable.	The specified <i>fOption</i> is not supported. Given specified <i>fOption</i> value, the value specified for the argument <i>vParam</i> is not supported.

Restrictions

For compatibility with ODBC applications, *fOption* values of SQL_CURRENT_QUALIFIER and SQL_PACKET_SIZE are also recognized, but not supported. If either of these two options are specified, SQL_ERROR is returned on the function call (SQLSTATE S1C00).

ODBC *fOption* values of SQL_TRANSLATE_DLL and SQL_TRANSLATE_OPTION are not supported since DB2 handles codepage conversion at the server, not the client.

Example

See "Example" on page 132.

References

- "SQLGetConnectOption - Returns current setting of a connect option" on page 202
- "SQLGetStmtOption - Returns current setting of a statement option" on page 273
- "SQLSetStmtOption - Set statement option" on page 367

SQLSetCursorName - Set cursor name

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

SQLSetCursorName() associates a cursor name with the statement handle. This function is optional since DB2 ODBC implicitly generates a cursor name when each statement handle is allocated.

Syntax

```
SQLRETURN SQLSetCursorName (SQLHSTMT      hstmt,
                             SQLCHAR       FAR *szCursor,
                             SQLSMALLINT   cbCursor);
```

Function arguments

Table 142. SQLSetCursorName arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle
SQLCHAR *	<i>szCursor</i>	input	Cursor name
SQLSMALLINT	<i>cbCursor</i>	input	Length of contents of <i>szCursor</i> 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 associated with positioned UPDATE and DELETE statements refer to the internal name.

Recommendation: Do not use SQLSetCursorName(). Instead, use the internal name which can be obtained by calling SQLGetCursorName().

Cursor names must follow these rules:

- All cursor names within the connection must be unique.
- Each cursor name must be less than or equal to 18 bytes in length. Any attempt to set a cursor name longer than 18 bytes results in truncation of that cursor name to 18 bytes. (No warning is generated.)
- Since internally generated names begin with SQLCUR, 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.
- Since a cursor name is considered an identifier in SQL, it must begin with an English letter (a-z, A-Z) followed by any combination of digits (0-9), English letters or the underscore character (_).
- To permit cursor names containing characters other than those listed above (such as National Language Set or Double Bytes Character Set characters), the application must enclose the cursor name in double quotes (").

SQLSetCursorName

- 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

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 143. SQLSetCursorName SQLSTATEs

SQLSTATE	Description	Explanation
34000	Invalid cursor name.	The cursor name specified by the argument <i>szCursor</i> is invalid. The cursor name either begins with SQLCUR, SQL_CUR, or SQLCURQRS or violates the cursor naming rules (Must begin with a-z or A-Z followed by any combination of English letters, digits, or the '_' character.
		The cursor name specified by the argument <i>szCursor</i> already exists.
		The cursor name length is greater than the value returned by SQLGetInfo() with the SQL_MAX_CURSOR_NAME_LEN argument.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY009	Invalid use of a null pointer.	<i>szCursor</i> is a null pointer.
HY010	Function sequence error.	There is an open or positioned cursor on the statement handle.
		The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation called prior to SQLSetCursorName().
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The argument <i>cbCursor</i> is less than 0, but not equal to SQL_NTS.

Restrictions

None.

Example

```

/* ... */
SQLCHAR      sqlstmt[] =
              "SELECT name, job FROM staff "
              "WHERE job='Clerk'  FOR UPDATE OF job";
/* ... */
/* allocate second statement handle for update statement */
rc2 = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt2);

/* Set Cursor for the SELECT statement's handle */
rc = SQLSetCursorName(hstmt1, "JOBBCURS", SQL_NTS);

rc = SQLExecDirect(hstmt1, sqlstmt, SQL_NTS);

/* bind name to first column in the result set */
rc = SQLBindCol(hstmt1, 1, SQL_C_CHAR, (SQLPOINTER) name.s, 10,
               &name.ind);

/* bind job to second column in the result set */
rc = SQLBindCol(hstmt1, 2, SQL_C_CHAR, (SQLPOINTER) job.s, 6,
               &job.ind);

printf("Job Change for all clerks\n");

while ((rc = SQLFetch(hstmt1)) == SQL_SUCCESS) {
    printf("Name: %-9.9s Job: %-5.5s \n", name.s, job.s);
    printf("Enter new job or return to continue\n");
    gets(newjob);
    if (newjob[0] != '\0') {
        sprintf(updstmt,
              "UPDATE staff set job = '%s' where current of JOBBCURS",
              newjob);
        rc2 = SQLExecDirect(hstmt2, updstmt, SQL_NTS);
    }
}
if (rc != SQL_NO_DATA_FOUND)
    check_error(henv, hdbc, hstmt1, rc, __LINE__, __FILE__);
/* ... */

```

References

- “SQLGetCursorName - Get cursor name” on page 204

SQLSetEnvAttr - Set environment attribute

Purpose

Specification:		X/OPEN CLI	ISO CLI
----------------	--	------------	---------

SQLSetEnvAttr() sets attributes that govern aspects of environments.

Syntax

```
SQLRETURN SQLSetEnvAttr (SQLHENV      EnvironmentHandle,
                        SQLINTEGER     Attribute,
                        SQLPOINTER     ValuePtr,
                        SQLINTEGER     StringLength);
```

Function arguments

Table 144. SQLSetEnvAttr arguments

Data type	Argument	Use	Description
SQLHENV	<i>EnvironmentHandle</i>	Input	Environment handle.
SQLINTEGER	<i>Attribute</i>	Input	Environment attribute to set. See Table 145 on page 351 for the list of attributes and their descriptions.
SQLPOINTER	<i>ValuePtr</i>	Input	The desired value for <i>Attribute</i> .
SQLINTEGER	<i>StringLength</i>	Input	Length of <i>ValuePtr</i> in bytes if the attribute value is a character string. If <i>Attribute</i> does not denote a string, DB2 ODBC ignores <i>StringLength</i> .

Usage

When set, the attributes value affects all connections in this environment.

The application can obtain the current attribute value by calling SQLGetEnvAttr().

Table 145 on page 351 lists the SQLSetEnvAttr() *Attribute* values. *Attribute* values were renamed in ODBC 3.0. For a summary of the *Attributes* renamed in ODBC 3.0, see Table 197 on page 502.

Table 145. Environment attributes

Attribute	Contents
SQL_ATTR_ODBC_VERSION	<p data-bbox="643 254 1456 342">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.</p> <p data-bbox="643 365 1313 394">The following values are used to set the value of this attribute:</p> <ul data-bbox="643 401 1456 867" style="list-style-type: none"> <li data-bbox="643 401 1456 632">• SQL_OV_ODBC3: Causes the following ODBC 3.0 behavior: <ul style="list-style-type: none"> <li data-bbox="670 436 1456 499">– DB2 ODBC returns and expects ODBC 3.0 data type codes for date, time, and timestamp. <li data-bbox="670 506 1456 569">– DB2 ODBC returns ODBC 3.0 SQLSTATE codes when SQLGetDiagRec(), SQLGetDiagField(), or SQLGetDiagRec() are called. <li data-bbox="670 575 1456 632">– The <i>CatalogName</i> argument in a call to SQLTables() accepts a search pattern. <li data-bbox="643 638 1456 867">• SQL_OV_ODBC2 causes the following ODBC 2.x behavior: <ul style="list-style-type: none"> <li data-bbox="670 674 1456 737">– DB2 ODBC returns and expects ODBC 2.x data type codes for date, time, and timestamp. <li data-bbox="670 743 1456 806">– DB2 ODBC returns ODBC 2.0 SQLSTATE codes when SQLGetDiagRec(), SQLGetDiagField(), or SQLGetDiagRec() are called. <li data-bbox="670 812 1456 867">– The <i>CatalogName</i> argument in a call to SQLTables() does <i>not</i> accept a search pattern.
SQL_ATTR_OUTPUT_NTS	<p data-bbox="643 879 1456 942">A 32-bit integer value which controls the use of null-termination in output arguments. The possible values are:</p> <ul data-bbox="643 949 1456 1108" style="list-style-type: none"> <li data-bbox="643 949 1456 1037">• SQL_TRUE: DB2 ODBC uses null termination to indicate the length of output character strings. This is the default. <li data-bbox="643 1043 1456 1108">• SQL_FALSE: DB2 ODBC does not use null termination in output character strings. <p data-bbox="643 1121 1456 1209">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.</p> <p data-bbox="643 1232 1456 1287">This attribute can only be set when there are no connection handles allocated under this environment.</p>

SQLSetEnvAttr

Table 145. Environment attributes (continued)

Attribute	Contents
SQL_ATTR_CONNECTTYPE	<p>A 32-bit integer value that specifies whether this application is to operate in a coordinated or uncoordinated distributed environment. The possible values are:</p> <ul style="list-style-type: none">• 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 <code>SQLEndTran()</code> and not all of the connections commit successfully, the application is responsible for recovery. This corresponds to <code>CONNECT</code> (type 1) semantics subject to the restrictions described in “DB2 ODBC restrictions on the ODBC connection model” on page 17. This is the default.• SQL_COORDINATED_TRANS: The application wishes to have commit and rollbacks coordinated among multiple database connections. In contrast to the <code>SQL_CONCURRENT_TRANS</code> setting described above, the application is permitted only one open connection per database. <p>This attribute must be set before allocating any connection handles, otherwise, the <code>SQLSetEnvAttr()</code> call is rejected.</p> <p>All the connections within an application must have the same <code>SQL_ATTR_CONNECTTYPE</code> and <code>SQL_ATTR_SYNCPOINT</code> values. This attribute can also be set using the <code>SQLSetConnectAttr</code> function.</p> <p>Recommendation: Have the application set the <code>SQL_ATTR_CONNECTTYPE</code> 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 <code>SQLSetConnectAttr()</code> as <code>SQLSetEnvAttr()</code> is not supported in ODBC.</p> <p>Note: This is an IBM-defined extension.</p>
SQL_ATTR_MAXCONN	<p>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.</p> <p>This can be used as a governor for the maximum number of connections on a per application basis.</p> <p>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.</p> <p>Recommendation: Have the application set <code>SQL_ATTR_MAXCONN</code> at the environment level rather than on a connection basis. ODBC applications must set this attribute at the connection level since <code>SQLSetEnvAttr()</code> is not supported in ODBC.</p> <p>Note: This is an IBM-defined extension.</p>

Return codes

- `SQL_SUCCESS`
- `SQL_INVALID_HANDLE`
- `SQL_ERROR`

Diagnostics

Table 146. SQLSetEnvAttr SQLSTATEs

SQLSTATE	Description	Explanation
HY009	Invalid use of a null pointer.	A null pointer was passed for <i>ValuePtr</i> and the value in <i>*ValuePtr</i> was a string value.
HY011	Operation invalid at this time.	Applications cannot set environment attributes while connection handles are allocated on the environment handle.
HY024	Invalid attribute value.	Given the specified <i>Attribute</i> value, an invalid value was specified in <i>*ValuePtr</i> .
HY090	Invalid string or buffer length.	The <i>StringLength</i> argument was less than 0, but was not SQL_NTS.
HY092	Option type out of range.	The value specified for the argument <i>Attribute</i> was not valid for this version of DB2 ODBC.
HYC00	Driver not capable.	The specified <i>Attribute</i> is not supported by DB2 ODBC. Given specified <i>Attribute</i> value, the value specified for the argument <i>ValuePtr</i> is not supported.

Restrictions

None.

Example

See also, “Distributed unit of work example” on page 395.

```
SQLINTEGER output_nts,autocommit;
rc = SQLSetEnvAttr( henv,
                   SQL_ATTR_OUTPUT_NTS,
                   ( SQLPOINTER ) output_nts,
                   0
                 );
CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc );
```

References

- “SQLAllocHandle - Allocate handle” on page 79
- “SQLGetEnvAttr - Returns current setting of an environment attribute” on page 226
- “SQLSetStmtAttr - Set options related to a statement” on page 360

SQLSetParam - Binds a parameter marker to a buffer

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	
-----------------------	-----------------	-------------------	--

Note: In ODBC 2.0, this function is replaced by `SQLBindParameter()`. See the restrictions section below for details.

`SQLSetParam()` is used to associate (bind) parameter markers in an SQL statement to one of the following:

- Application variables (storage buffers), for all data types. In this case data is transferred from the application to the DBMS when `SQLExecute()` or `SQLExecDirect()` is called. Data conversion can occur as the data is transferred.
- A LOB locator, for SQL LOB data types. In this case, the application transfers a LOB locator value (not the LOB data itself) to the server when the SQL statement is executed.

Syntax

```
SQLRETURN SQLSetParam (SQLHSTMT hstmt,
                        SQLUSMALLINT ipar,
                        SQLSMALLINT fCType,
                        SQLSMALLINT fSqlType,
                        SQLUINTEGER cbParamDef,
                        SQLSMALLINT ibScale,
                        SQLPOINTER rgbValue,
                        SQLINTEGER FAR *pcbValue);
```

Function arguments

Table 147. *SQLSetParam* arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>hstmt</i>	input	Statement handle.
SQLUSMALLINT	<i>ipar</i>	input	Parameter marker number, ordered sequentially left to right, starting at 1.

Table 147. SQLSetParam arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	<i>fCType</i>	input	<p>C data type of argument. The following types are supported:</p> <ul style="list-style-type: none"> • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT • SQL_C_WCHAR <p>Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type for the type indicated in <i>fSqlType</i>. See Table 4 on page 31 for more information.</p>
SQLSMALLINT	<i>fSqlType</i>	input	<p>SQL data type of column. The supported types are:</p> <ul style="list-style-type: none"> • SQL_BINARY • SQL_BLOB • SQL_BLOB_LOCATOR • SQL_CHAR • SQL_CLOB • SQL_CLOB_LOCATOR • SQL_DBCLOB • SQL_DBCLOB_LOCATOR • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_ROWID • SQL_SMALLINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC <p>Note: 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.</p>

SQLSetParam

Table 147. SQLSetParam arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER	<i>cbParamDef</i>	input	<p>Precision of the corresponding parameter marker. If <i>fSqlType</i> denotes:</p> <ul style="list-style-type: none">• A binary or single byte character string (for example, SQL_CHAR, SQL_BINARY), this is the maximum length in bytes for this parameter marker.• A double byte character string (for example, SQL_GRAPHIC), this is the maximum length in double-byte characters for this parameter.• SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision.• SQL_ROWID, this must be set to 40, the maximum length in bytes for this data type. Otherwise, an error is returned.• Otherwise, this argument is ignored.
SQLSMALLINT	<i>ibScale</i>	input	<p>Scale of the corresponding parameter marker if <i>fSqlType</i> is SQL_DECIMAL or SQL_NUMERIC. If <i>fSqlType</i> is SQL_TYPE_TIMESTAMP, this is the number of digits to the right of the decimal point in the character representation of a timestamp (for example, the scale of yyyy-mm-dd hh:mm:ss.fff is 3).</p> <p>Other than for the <i>fSqlType</i> values mentioned here, <i>ibScale</i> is ignored.</p>
SQLPOINTER	<i>rgbValue</i>	input (deferred)	<p>Pointer to the location which contains (when the statement is executed) the actual values for the associated parameter marker.</p>
SQLINTEGER *	<i>pcbValue</i>	input (deferred)	<p>Pointer to the location which contains (when the statement is executed) the length of the parameter marker value stored at <i>rgbValue</i>.</p> <p>To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA.</p> <p>If <i>fCType</i> is SQL_C_CHAR, this storage location must contain either the exact length of the data stored at <i>rgbValue</i>, or SQL_NTS if the contents at <i>rgbValue</i> are null-terminated.</p> <p>If <i>fCType</i> indicates character data (explicitly, or implicitly using SQL_C_DEFAULT), and this pointer is set to NULL, it is assumed that the application always provides a null-terminated string in <i>rgbValue</i>. This also implies that this parameter marker never contains a null value.</p> <p>If <i>fSqlType</i> indicates a graphic data type, and the <i>fCType</i> is SQL_C_CHAR, the pointer to <i>pcbValue</i> can never be NULL and the contents of <i>pcbValue</i> can never hold SQL_NTS. In general for graphic data types, this length should be the number of octets that the double byte data occupies; therefore, the length should always be a multiple of 2. In fact, if the length is odd, then an error occurs when the statement is executed.</p>

Usage

A parameter marker is represented by a "?" character in an SQL statement and is used to indicate a position in the statement where an application supplied value is to be substituted when the statement is executed. This value can be obtained from:

- An application variable.
SQLSetParam() (or SQLBindParameter()) is used to bind the application storage area to the parameter marker.
- A LOB value from the database server (by specifying a LOB locator).
SQLSetParam() (or SQLBindParameter()) is used to bind a LOB locator to the parameter marker. The LOB value itself is supplied by the database server, so only the LOB locator is transferred between the database server and the application.
An application can use a locator with SQLGetSubString(), SQLGetSubString() or SQLGetLength().SQLGetSubString() can either return another locator, or the data itself. All locators remain valid until the end of the transaction in which they are created (even when the cursor moves to another row, or until it is freed using the FREE LOCATOR statement.

The application must bind a variable to each parameter marker in the SQL statement before executing the SQL statement. For this function, *rgbValue* and *pcbValue* are deferred arguments. The storage locations must be valid and contain input data values when the statement is executed. This means either keeping the SQLExecDirect() or SQLExecute() call in the same procedure scope as the SQLBindParameter() calls, or, these storage locations must be dynamically allocated or declared statically or globally.

SQLSetParam() can be called before SQLPrepare() if the columns in the result set are known, otherwise the attributes of the result set can be obtained after the statement is prepared.

Parameter markers are referenced by number (*icol*) and are numbered sequentially from left to right, starting at 1.

All parameters bound by this function remain in effect until SQLFreeHandle() is called with *HandleType* set to SQL_HANDLE_STMT or SQLFreeStmt() is called with the SQL_RESET_PARAMS option, or until SQLSetParam() is called again for the same parameter *ipar* number.

After the SQL statement is executed, and the results processed, the application can reuse the statement handle to execute a different SQL statement. If the parameter marker specifications are different (number of parameters, length or type), then SQLFreeStmt() should be called with SQL_RESET_PARAMS to reset or clear the parameter bindings.

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

If *fSqlType* is SQL_ROWID, the *cbParamDef* value must be set to 40, the maximum length in bytes for a ROWID data type. If *cbParamDef* is not set to 40, the application receives SQLSTATE=22001 when *cbParamDef* is less than 40 and SQLSTATE=HY104 when *cbParamDef* is greater than 40.

An application can pass the value for a parameter either in the *rgbValue* buffer or with one or more calls to SQLPutData(). In the latter case, 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.

SQLSetParam

Since 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.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 148. *SQLSetParam* SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The conversion from the data value identified by the <i>fCType</i> argument to the data type identified by the <i>fSqlType</i> argument is not a meaningful conversion. (For example, conversion from <code>SQL_C_TYPE_DATE</code> to <code>SQL_DOUBLE</code> .)
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY003	Program type out of range.	The value specified by the argument <i>fCType</i> is not a valid data type or <code>SQL_C_DEFAULT</code> .
HY004	Invalid SQL data type.	The value specified for the argument <i>fSqlType</i> is not a valid SQL data type.
HY009	Invalid use of a null pointer.	The argument <i>rgbValue</i> is a null pointer.
HY010	Function sequence error.	There is an open or positioned cursor on the statement handle. The function is called while in a data-at-execute (<code>SQLParamData()</code> , <code>SQLPutData()</code>) operation, called prior to <code>SQLSetCursorName()</code> .
HY013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.
HY104	Invalid precision or scale value.	The value specified for <i>cbParamDef</i> is either less than 0 or greater than the permissible range for the <i>fSqlType</i> . The value specified for <i>fSqlType</i> is either <code>SQL_DECIMAL</code> or <code>SQL_NUMERIC</code> and the value specified for <i>ibScale</i> is less than 0 or greater than the value for the argument <i>cbParamDef</i> (precision). The value specified for <i>fCType</i> is <code>SQL_C_TYPE_TIMESTAMP</code> and the value for <i>fSqlType</i> is either <code>SQL_CHAR</code> or <code>SQL_VARCHAR</code> and the value for <i>ibScale</i> is less than 0 or greater than 6.
HYC00	Driver not capable.	DB2 ODBC or the data source does not support the conversion specified by the combination of the value specified for the argument <i>fCType</i> and the value specified for the argument <i>fSqlType</i> . The value specified for the argument <i>fCType</i> or <i>fSqlType</i> is not supported by either DB2 ODBC or the data source.
S1093	Invalid parameter number.	The value specified for the argument <i>ipar</i> is less than 1 or greater than the maximum number of parameters supported by the server.

Restrictions

In ODBC 2.0, `SQLSetParam()` has replaced by `SQLBindParameter()`.

`SQLSetParam()` cannot be used to:

- Bind application variables to parameter markers in a stored procedure CALL statement.
- Bind arrays of application variables when `SQLParamOptions()` has been used to specify multiple input parameter values.

`SQLBindParameter()` should be used instead in both of the above situations.

Example

See “Example” on page 303.

References

- “SQLBindParameter - Binds a parameter marker to a buffer or LOB locator” on page 91
- “SQLExecDirect - Execute a statement directly” on page 161
- “SQLExecute - Execute a statement” on page 166
- “SQLPrepare - Prepare a statement” on page 300

SQLSetStmtAttr - Set options related to a statement

Purpose

Specification:	ODBC 3.0	X/OPEN CLI	ISO CLI
----------------	----------	------------	---------

SQLSetStmtAttr() sets options related to a statement. To set an option for all statements associated with a specific connection, an application can call SQLSetConnectAttr().

Syntax

```
SQLRETURN SQLSetStmtAttr (SQLHSTMT StatementHandle,
                          SQLINTEGER Attribute,
                          SQLPOINTER ValuePtr,
                          SQLINTEGER StringLength);
```

Function arguments

Table 149. SQLSetStmtAttr arguments

Data type	Argument	Use	Description
SQLHSTMT	<i>StatementHandle</i>	input	Statement handle.
SQLINTEGER	<i>Attribute</i>	input	Statement attribute to set. Refer to Table 150 on page 361 for a complete list of attributes.
SQLPOINTER	<i>ValuePtr</i>	input	Pointer to the value to be associated with <i>Attribute</i> . Depending on the value of <i>Attribute</i> , <i>*ValuePtr</i> will be a 32-bit unsigned integer value or point to a null-terminated character string. If the <i>Attribute</i> argument is a driver-specific value, the value in <i>*ValuePtr</i> might be a signed integer.
SQLINTEGER	<i>StringLength</i>	input	Information about the <i>*ValuePtr</i> argument. <ul style="list-style-type: none"> For ODBC-defined attributes: <ul style="list-style-type: none"> If <i>ValuePtr</i> points to a character string, this argument should be the length of <i>*ValuePtr</i>. If <i>ValuePtr</i> points to an integer, <i>BufferLength</i> is ignored. For driver-defined attributes (IBM extension): <ul style="list-style-type: none"> If <i>ValuePtr</i> points to a character string, this argument should be the length of <i>*ValuePtr</i> or SQL_NTS if it is a null-terminated string. If <i>ValuePtr</i> points to an integer, <i>BufferLength</i> 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 option 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 01S02 (option 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 null-terminated character string or a 32-bit integer value. The format of each *ValuePtr* value is noted in the attributes description shown in Table 150. This format applies to the information returned for each attribute in `SQLGetStmtAttr()`. Character strings that the *ValuePtr* argument of `SQLSetStmtAttr()` point to have a length of *StringLength*.

DB2 ODBC supports all of the ODBC 2.0 *Attribute* values that are renamed in ODBC 3.0. For a summary of the *Attribute* values renamed in ODBC 3.0, see Table 198 on page 502.

Overriding DB2 CCSIDs from DSNHDECP: DB2 ODBC extensions to `SQLSetStmtAttr()` and `SQLSetStmtAttr()` allow an application to override the CCSID settings of the DB2 subsystem to which they are currently attached. This extension is intended for applications that are attempting to send and receive data to and from DB2 in a CCSID that differs from the default settings in the DB2 DSNHDECP.

The CCSID override applies only to input data bound to parameter markers through `SQLBindParameter()` and output data bound to columns through `SQLBindCol()`.

The CCSID override applies on a statement level only. DB2 will continue to use the default CCSID settings in the DB2 DSNHECP after the statement is dropped or if `SQL_CCSID_DEFAULT` is specified.

You can use `SQLGetStmtAttr()` and `SQLGetStmtAttr()` to query the settings of the current statement handle CCSID override.

Table 150. Statement attributes

Attribute	ValuePtr contents
Note: Values shown in bold are default values.	
SQL_ATTR_BIND_TYPE SQL_ATTR_ROW_BIND_TYPE	<p>A 32-bit integer value that sets the binding orientation to be used when <code>SQLExtendedFetch()</code> is called with this statement handle. <i>Column-wise binding</i> is selected by supplying the value SQL_BIND_BY_COLUMN for the argument <i>vParam</i>. <i>Row-wise binding</i> is selected by supplying a value for <i>vParam</i> specifying the length of the structure or an instance of a buffer into which result columns are bound.</p> <p>For row-wise binding, the length specified in <i>vParam</i> 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 <code>sizeof</code> operator with structures or unions in ANSI C, this behavior is guaranteed.)</p>

SQLSetStmtAttr

Table 150. Statement attributes (continued)

Attribute	ValuePtr contents
SQL_ATTR_CLOSE_BEHAVIOR	<p>A 32-bit integer that forces the release of locks upon an underlying CLOSE CURSOR operation. The possible values are:</p> <ul style="list-style-type: none">• SQL_CC_NO_RELEASE: locks are not released when the cursor on this statement handle is closed.• SQL_CC_RELEASE: locks are released when the cursor on this statement handle is closed. <p>Typically cursors are explicitly closed when the function SQLFreeStmt() is called with the SQL_CLOSE option 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).</p>
SQL_ATTR_CONCURRENCY	<p>If specified, DB2 ODBC returns S1C00 on SQLSetConnectOption and HY011 on SQLGetConnectOption.</p>
SQL_ATTR_CURSOR_HOLD	<p>A 32-bit integer which specifies whether the cursor associated with this hstmt is preserved in the same position as before the COMMIT operation, and whether the application can fetch without executing the statement again.</p> <ul style="list-style-type: none">• SQL_CURSOR_HOLD_ON• SQL_CURSOR_HOLD_OFF <p>The default value when an hstmt is first allocated is SQL_CURSOR_HOLD_ON.</p> <p>This option cannot be specified while there is an open cursor on this hstmt.</p>
SQL_ATTR_CURSOR_TYPE	<p>A 32-bit integer value that specifies the cursor type. The currently supported value is:</p> <ul style="list-style-type: none">• SQL_CURSOR_FORWARD_ONLY - Cursor behaves as a forward only scrolling cursor. <p>This option cannot be set if there is an open cursor on the associated <i>hstmt</i>.</p> <p>Note: ODBC architecture has also defined the following values, which are not supported by DB2 ODBC:</p> <ul style="list-style-type: none">• SQL_CURSOR_STATIC - The data in the result set appears to be static.• SQL_CURSOR_KEYSET_DRIVEN - The keys for the number of rows specified in the SQL_KEYSET_SIZE option is stored. DB2 ODBC does not support this option value.• SQL_CURSOR_DYNAMIC - The keys for the rows in the rowset are saved. DB2 ODBC does not support this option value. <p>If one of these values is used, SQL_SUCCESS_WITH_INFO (SQLSTATE 01S02) is returned and the value remains unchanged.</p>
SQL_ATTR_MAX_LENGTH	<p>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 01004 (data truncated). The default value for <i>vParam</i> is 0; 0 means that DB2 ODBC attempts to return all available data for character or binary type data.</p>

Table 150. Statement attributes (continued)

Attribute	ValuePtr contents
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 <i>vParam</i> is 0 ; 0 means all rows are returned.
SQL_ATTR_NODESCRIBE	<p>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 <code>SQLSetColAttributes()</code>.</p> <ul style="list-style-type: none"> • SQL_NODESCRIBE_OFF • SQL_NODESCRIBE_ON <p>This option cannot be specified while there is an open cursor on this hstmt.</p> <p>This option is used in conjunction with the function <code>SQLSetColAttributes()</code> by an application which has prior knowledge of the exact nature of the result set to be returned and which does not wish to incur the extra network traffic associated with the descriptor information needed by DB2 ODBC to provide client side processing.</p> <p>Note: This option is an IBM-defined extension.</p>
SQL_ATTR_NOSCAN	<p>A 32-bit integer value that specifies whether DB2 ODBC will scan SQL strings for escape clauses. The two permitted values are:</p> <ul style="list-style-type: none"> • SQL_NOSCAN_OFF - SQL strings are scanned for escape clause sequences. • SQL_NOSCAN_ON - SQL strings are not scanned for escape clauses. Everything is sent directly to the server for processing. <p>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.</p>
SQL_ATTR_RETRIEVE_DATA	<p>A 32-bit integer value indicating whether DB2 ODBC should actually retrieve data from the database when <code>SQLExtendedFetch()</code> is called. The possible values are:</p> <ul style="list-style-type: none"> • SQL_RD_ON: <code>SQLExtendedFetch()</code> does retrieve data. • SQL_RD_OFF: <code>SQLExtendedFetch()</code> does not retrieve data. This is useful for verifying whether rows exist without incurring the overhead of sending long data from the database server. DB2 ODBC internally retrieves all the fixed length columns, such as <code>INTEGER</code> and <code>SMALLINT</code>; so there is still some overhead. <p>This option cannot be set if the cursor is open.</p>
SQL_ATTR_ROW_ARRAY_SIZE	A 32-bit integer value that specifies the number of rows in the rowset. This is the number of rows returned by each call to <code>SQLExtendedFetch()</code> . The default value is 1 which is equivalent to making a single <code>SQLFetch()</code> call. This option can be specified for an open cursor and becomes effective on the next <code>SQLExtendedFetch()</code> call.
SQL_ATTR_ROWSET_SIZE	A 32-bit integer value that specifies the number of rows in the rowset. A rowset is the array of rows returned by each call to <code>SQLExtendedFetch()</code> . The default value is 1 , which is equivalent to making a single <code>SQLFetch()</code> . This option can be specified even when the cursor is open and becomes effective on the next <code>SQLExtendedFetch()</code> call.

SQLSetStmtAttr

Table 150. Statement attributes (continued)

Attribute	ValuePtr contents
SQL_ATTR_STMTTXN_ISOLATION SQL_ATTR_TXN_ISOLATION	<p data-bbox="711 258 1398 369">A 32-bit integer value that sets the transaction isolation level for the current hstmt. This overrides the default value set at the connection level (refer also to “SQLSetConnectOption - Set connection option” on page 345 for the permitted values).</p> <p data-bbox="711 396 1344 451">This option cannot be set if there is an open cursor on this statement handle (SQLSTATE 24000).</p> <p data-bbox="711 478 1382 533">The value SQL_ATTR_STMTTXN_ISOLATION is synonymous with SQL_ATTR_TXN_ISOLATION.</p> <p data-bbox="711 537 1386 590">Note: It is an IBM extension to allow setting this option at the statement level.</p>
SQL_CCSID_CHAR	<p data-bbox="711 611 1377 665">A 32-bit integer value that specifies the CCSID of input/output data, to or from a column of the following SQL data types:</p> <ul data-bbox="711 669 987 751" style="list-style-type: none"><li data-bbox="711 669 870 693">• SQL_CHAR<li data-bbox="711 697 919 720">• SQL_VARCHAR<li data-bbox="711 724 987 747">• SQL_LONGVARCHAR <p data-bbox="711 779 1425 890">This CCSID will override the default CCSID setting from DB2 DSNHECP. The input data should be bound to parameter markers through SQLBindParameter(). The output data should be bound to columns through SQLBindCol().</p> <p data-bbox="711 919 1382 999">SQL_CCSID_DEFAULT is the default value of this statement option, therefore, the CCSIDs from the DB2 DSNHECP will be used.</p>

Table 150. Statement attributes (continued)

Attribute	ValuePtr contents
SQL_CCSID_GRAPHIC	<p>A 32-bit integer value that specifies the CCSID of input/output data, to or from a column of the following SQL data types:</p> <ul style="list-style-type: none"> • SQL_GRAPHIC • SQL_VARGRAPHIC • SQL_LONGVARGRAPHIC <p>This CCSID will override the default CCSID setting from DB2 DSNHECP. The input data should be bound to parameter markers through SQLBindParameter(). The output data should be bound to columns through SQLBindCol().</p> <p>SQL_CCSID_DEFAULT is the default value of this statement option, therefore, the CCSIDs from the DB2 DSNHECP will be used.</p> <p>Note: DB2 UDB for OS/390 ODBC extensions to SQLSetStmtAttr() and SQLSetStmtAttr() allow an application to override the CCSID settings of the DB2 subsystem to which they are currently attached. This extension is intended for applications that are attempting to send and receive data to and from DB2 in a CCSID that differs from the default settings in the DB2 DSNHECP.</p> <p>The CCSID override applies only to input data bound to parameter markers through SQLBindParameter() and output data bound to columns through SQLBindCol().</p> <p>The CCSID override applies on a statement level only. DB2 will continue to use the default CCSID settings in the DB2 DSNHECP after the statement is dropped or if SQL_CCSID_DEFAULT is specified.</p> <p>You can use SQLGetStmtAttr() and SQLGetStmtAttr() to query the settings of the current statement handle CCSID override.</p>

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

Table 151. SQLSetStmtAttr SQLSTATES

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed.	DB2 did not support the value specified in *ValuePtr, or the value specified in *ValuePtr was invalid due to SQL constraints or requirements. Therefore, DB2 ODBC substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Unable to connect to data source.	The communication link between the application and the data source failed before the function completed.

SQLSetStmtAttr

Table 151. SQLSetStmtAttr SQLSTATEs (continued)

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	The <i>Attribute</i> was SQL_ATTR_CONCURRENCY and the cursor was open.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by SQLGetDiagRec() in the <i>*MessageText</i> buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 ODBC was not able to allocate memory for the specified handle.
HY009	Invalid use of a null pointer.	A null pointer was passed for <i>ValuePtr</i> and the value in <i>*ValuePtr</i> was a string value.
HY010	Function sequence error.	SQLExecute() or SQLExecDirect() was called with the statement handle, and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. Invoke SQLCancel() to cancel the data-at-execution condition.
HY011	Operation invalid at this time.	The <i>Attribute</i> was SQL_ATTR_CONCURRENCY and the statement was prepared.
HY024	Invalid attribute value.	Given the specified <i>Attribute</i> value, an invalid value was specified in <i>*ValuePtr</i> .
HY090	Invalid string or buffer length.	The <i>StringLength</i> argument was less than 0, but was not SQL_NTS.
HY092	Option type out of range.	The value specified for the argument <i>Attribute</i> was not valid for this version of DB2 ODBC.
HYC00	Driver not capable.	The value specified for the argument <i>Attribute</i> was a valid connection or statement attribute for the version of the DB2 ODBC driver, but was not supported by the data source.

Restrictions

None.

Example

```
rc = SQLSetStmtAttr( hstmt,  
                    SQL_ATTR_CURSOR_HOLD,  
                    ( void * ) SQL_CURSOR_HOLD_OFF,  
                    0 ) ;  
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
```

References

- “SQLGetConnectAttr - Get current attribute setting” on page 199
- “SQLSetConnectAttr - Set connection attributes” on page 336
- “SQLGetStmtAttr - Get current setting of a statement attribute” on page 270
- “SQLCancel - Cancel statement” on page 102

SQLSetStmtOption - Set statement option

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	
----------------	----------	------------	--

In ODBC 3.0, `SQLSetStmtAttr()` replaces the ODBC 2.0 function `SQLSetStmtOption()`. See `SQLSetStmtAttr()` for more information.

`SQLSetStmtOption()` sets an attribute of a specific statement handle. To set an option for all statement handles associated with a connection handle, the application can call `SQLSetConnectOption()` (see “`SQLSetConnectOption - Set connection option`” on page 345).

Syntax

```
SQLRETURN SQLSetStmtOption (SQLHSTMT      hstmt,
                             SQLUSMALLINT  fOption,
                             SQLINTEGER    vParam);
```

Function arguments

Table 152. `SQLSetStmtOption` arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLUSMALLINT	fOption	input	Option to set.
SQLINTEGER	vParam	input	Value associated with <i>fOption</i> . <i>vParam</i> can be a 32-bit integer value or a pointer to a null-terminated string.

Usage

Statement options for an *hstmt* remain in effect until they are changed by another call to `SQLSetStmtOption()` or `SQLSetConnectOption()`, or the *hstmt* is dropped by calling `SQLFreeStmt()` with the `SQL_DROP` option. Calling `SQLFreeStmt()` with the `SQL_CLOSE`, `SQL_UNBIND`, or `SQL_RESET_PARAMS` options does not reset statement options.

The format of *vParam* depends on the value specified *fOption*. The format of each is noted in Table 150 on page 361. If the format denotes a pointer to a null-terminated character string the maximum length is `SQL_MAX_OPTION_STRING_LENGTH` (excluding the null terminator).

Note: Currently no statement option requires a string.

Return codes

- `SQL_SUCCESS`
- `SQL_ERROR`
- `SQL_INVALID_HANDLE`

SQLSetStmtOption

Diagnostics

Table 153. SQLSetStmtOption SQLSTATES

SQLSTATE	Description	Explanation
01S02	Option value changed.	A recognized concurrency value is specified for the SQL_CONCURRENCY option, but is not supported by DB2 ODBC.
24000	Invalid cursor state.	<i>fOption</i> is set to SQL_CONCURRENCY, SQL_CURSOR_TYPE, SQL_STMTTXN_ISOLATION, or SQL_TXN_ISOLATION and a cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1000	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 <i>szErrorMsg</i> describes the error and its cause.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	Given the specified <i>fOption</i> value, an invalid value is specified for the argument <i>vParam</i> .
S1010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation; called prior to SQLSetCursorName().
S1011	Operation invalid at this time.	The <i>fOption</i> is SQL_CONCURRENCY, SQL_CURSOR_HOLD, SQL_NODESCRIBE, SQL_RETRIEVE_DATA, SQL_(STMT)TXN_ISOLATION, or SQL_CURSOR_TYPE and the statement is prepared.
S1092	Option type out of range.	An invalid <i>fOption</i> value is specified.
S1C00	Driver not capable.	The option or option value is not supported.

Restrictions

ODBC also defines statement options SQL_KEYSET_SIZE, SQL_BOOKMARKS and SQL_SIMULATE_CURSOR. These options are not supported by DB2 ODBC. If either one is specified, SQL_ERROR (SQLSTATE S1C00) is returned.

Example

See “Example” on page 335.

References

- “SQLColAttributes - Get column attributes” on page 114
- “SQLExtendedFetch - Extended fetch (fetch array of rows)” on page 169
- “SQLFetch - Fetch next row” on page 176
- “SQLGetConnectOption - Returns current setting of a connect option” on page 202
- “SQLGetData - Get data from a column” on page 210
- “SQLGetStmtOption - Returns current setting of a statement option” on page 273
- “SQLParamOptions - Specify an input array for a parameter” on page 298
- “SQLSetConnectOption - Set connection option” on page 345

SQLSpecialColumns - Get special (row identifier) columns

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	
----------------	----------	------------	--

SQLSpecialColumns() returns unique row identifier information (primary key or unique index) for a table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN SQLSpecialColumns(SQLHSTMT hstmt,
                             SQLUSMALLINT fColType,
                             SQLCHAR FAR *szCatalogName,
                             SQLSMALLINT cbCatalogName,
                             SQLCHAR FAR *szSchemaName,
                             SQLSMALLINT cbSchemaName,
                             SQLCHAR FAR *szTableName,
                             SQLSMALLINT cbTableName,
                             SQLUSMALLINT fScope,
                             SQLUSMALLINT fNullable);
```

Function arguments

Table 154. SQLSpecialColumns arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLUSMALLINT	fColType	Input	Type of unique row identifier to return. Only the following type is supported: <ul style="list-style-type: none"> SQL_BEST_ROWID Returns the optimal set of columns that can uniquely identify any row in the specified table. <p>Note: For compatibility with ODBC applications, SQL_ROWVER is also recognized, but not supported; therefore, if SQL_ROWVER is specified, an empty result is returned.</p>
SQLCHAR *	szCatalogName	Input	Catalog qualifier of a 3 part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	cbCatalogName	Input	Length of <i>szCatalogName</i> . This must be a set to 0.
SQLCHAR *	szSchemaName	Input	Schema qualifier of the specified table.
SQLSMALLINT	cbSchemaName	Input	Length of <i>szSchemaName</i> .
SQLCHAR *	szTableName	Input	Table name.
SQLSMALLINT	cbTableName	Input	Length of <i>cbTableName</i> .

SQLSpecialColumns

Table 154. SQLSpecialColumns arguments (continued)

Data type	Argument	Use	Description
SQLUSMALLINT	fScope	Input	<p>Minimum required duration for which the unique row identifier is valid.</p> <p><i>fScope</i> must be one of the following:</p> <ul style="list-style-type: none">• 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 was updated or deleted by another transaction.• SQL_SCOPE_TRANSACTION: The row identifier is guaranteed to be valid for the duration of the current transaction. Note: This option is only valid if SQL_TXN_SERIALIZABLE and SQL_TXN_REPEATABLE_READ isolation options are set.• SQL_SCOPE_SESSION: The row identifier is guaranteed to be valid for the duration of the connection. Note: This option is not supported by DB2 for OS/390 and z/OS. <p>The duration over which a row identifier value is guaranteed to be valid depends on the current transaction isolation level. For information and scenarios involving isolation levels, see <i>DB2 SQL Reference</i>.</p>
SQLUSMALLINT	fNullable	Input	<p>Determines whether to return special columns that can have a NULL value.</p> <p>Must be one of the following:</p> <ul style="list-style-type: none">• SQL_NO_NULLS - The row identifier column set returned cannot have any NULL values.• SQL_NULLABLE - The row identifier column set returned can include columns where NULL values are permitted.

Usage

If multiple ways exist to uniquely identify any row in a table (that is, if there are multiple unique indexes on the specified table), then DB2 ODBC returns the *best* set of row identifier column sets based on its internal criterion.

If there is no column set that 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 155 on page 371 shows the order of the columns in the result set returned by `SQLSpecialColumns()`, sorted by SCOPE.

Since calls to `SQLSpecialColumns()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call

SQLGetInfo() with the SQL_MAX_COLUMN_NAME_LEN to determine the actual length of the COLUMN_NAME column supported by the connected DBMS.

Although new columns might be added and the names of the columns changed in future releases, the position of the current columns does not change.

Table 155. Columns returned by SQLSpecialColumns

Column number/name	Data type	Description
1 SCOPE	SMALLINT	<p>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 <i>fScope</i> argument: Actual scope of the row identifier. Contains one of the following values:</p> <ul style="list-style-type: none"> • SQL_SCOPE_CURROW • SQL_SCOPE_TRANSACTION • SQL_SCOPE_SESSION <p>See <i>fScope</i> in Table 154 on page 369 for a description of each value.</p>
2 COLUMN_NAME	VARCHAR(128) NOT NULL	Name of the column that is (or part of) the table's primary key.
3 DATA_TYPE	SMALLINT NOT NULL	SQL data type of the column. One of the values in the Symbolic SQL Data Type column in Table 4 on page 31.
4 TYPE_NAME	VARCHAR(128) NOT NULL	DBMS character string represented of the name associated with DATA_TYPE column value.
5 COLUMN_SIZE	INTEGER	<p>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.</p> <p>For date, time, timestamp data types, this is the total number of bytes required to display the value when converted to character.</p> <p>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.</p> <p>See Table 174 on page 486.</p>
6 BUFFER_LENGTH	INTEGER	<p>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 null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.</p> <p>See Table 176 on page 488.</p>
7 DECIMAL_DIGITS	SMALLINT	The scale of the column. NULL is returned for data types where scale is not applicable. See Table 175 on page 487.

SQLSpecialColumns

Table 155. Columns returned by SQLSpecialColumns (continued)

Column number/name	Data type	Description
8 PSEUDO_COLUMN	SMALLINT	Indicates whether or not the column is a pseudo-column. DB2 ODBC only returns: <ul style="list-style-type: none">• SQL_PC_NOT_PSEUDO DB2 DBMSs do not support pseudo columns. ODBC applications can receive the following values from other non-IBM RDBMS servers: <ul style="list-style-type: none">• SQL_PC_UNKNOWN• SQL_PC_PSEUDO

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 156. SQLSpecialColumns SQLSTATES

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the length arguments is less than 0, but not equal to SQL_NTS. The value of one of the length arguments exceeded the maximum length supported by the DBMS for that qualifier or name.
HY097	Column type out of range.	An invalid <i>fColType</i> value is specified.
HY098	Scope type out of range.	An invalid <i>fScope</i> value is specified.
HY099	Nullable type out of range.	An invalid <i>fNullable</i> values is specified.
HYC00	Driver not capable.	DB2 ODBC does not support <i>catalog</i> as a qualifier for table name.

Restrictions

None.

Example

```

/* ... */
SQLRETURN
list_index_columns(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */
    rc = SQLSpecialColumns(hstmt, SQL_BEST_ROWID, NULL, 0, schema, SQL_NTS,
                          tablename, SQL_NTS, SQL_SCOPE_CURROW, SQL_NULLABLE);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) column_name.s, 129,
                   &column_name.ind);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) type_name.s, 129,
                   &type_name.ind);

    rc = SQLBindCol(hstmt, 5, SQL_C_LONG, (SQLPOINTER) & precision,
                   sizeof(precision), &precision_ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_SHORT, (SQLPOINTER) & scale,
                   sizeof(scale), &scale_ind);

    printf("Primary Key or Unique Index for %s.%s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("  %s, %s ", column_name.s, type_name.s);
        if (precision_ind != SQL_NULL_DATA) {
            printf(" (%ld", precision);
        } else {
            printf("\n");
        }
        if (scale_ind != SQL_NULL_DATA) {
            printf(", %d)\n", scale);
        } else {
            printf(")\n");
        }
    }
}
/* ... */

```

References

- “SQLColumns - Get column information for a table” on page 124
- “SQLStatistics - Get index and statistics information for a base table” on page 374
- “SQLTables - Get table information” on page 382

SQLStatistics - Get index and statistics information for a base table

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	
-----------------------	-----------------	-------------------	--

SQLStatistics() retrieves index information for a given table. It also returns the cardinality and the number of pages associated with the table and the indexes on the table. The information is returned in a result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN SQLStatistics (SQLHSTMT hstmt,
                        SQLCHAR FAR *szCatalogName,
                        SQLSMALLINT cbCatalogName,
                        SQLCHAR FAR *szSchemaName,
                        SQLSMALLINT cbSchemaName,
                        SQLCHAR FAR *szTableName,
                        SQLSMALLINT cbTableName,
                        SQLUSMALLINT fUnique,
                        SQLUSMALLINT fAccuracy);
```

Function arguments

Table 157. SQLStatistics arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLCHAR *	szCatalogName	Input	Catalog qualifier of a 3 part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	cbCatalogName	Input	Length of <i>cbCatalogName</i> . This must be set to 0.
SQLCHAR *	szSchemaName	Input	Schema qualifier of the specified table.
SQLSMALLINT	cbSchemaName	Input	Length of <i>szSchemaName</i> .
SQLCHAR *	szTableName	Input	Table name.
SQLSMALLINT	cbTableName	Input	Length of <i>cbTableName</i> .
SQLUSMALLINT	fUnique	Input	Type of index information to return: <ul style="list-style-type: none"> SQL_INDEX_UNIQUE <p>Only unique indexes are returned.</p> <ul style="list-style-type: none"> SQL_INDEX_ALL <p>All indexes are returned.</p>
SQLUSMALLINT	fAccuracy	Input	Indicate whether the CARDINALITY and PAGES columns in the result set contain the most current information: <ul style="list-style-type: none"> 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. <p>Recommendation: New applications should not use this value.</p> <ul style="list-style-type: none"> 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 it is available):
 - when the TYPE column in the table below is set to SQL_TABLE_STAT, the number of rows in the table and the number of pages used to store the table.
 - when the TYPE column indicates an index, the number of unique values in the index, and the number of pages used to store the indexes.
- Information about each index, where each index column is represented by one row of the result set. The result set columns are given in Table 158 in the order shown; the rows in the result set are ordered by NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME and ORDINAL_POSITION.

Since calls to SQLStatistics() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 158. Columns returned by SQLStatistics

Column number/name	Data type	Description
1 TABLE_CAT	VARCHAR(128)	The 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 table.
4 NON_UNIQUE	SMALLINT	Indicates whether the index prohibits duplicate values: <ul style="list-style-type: none"> • SQL_TRUE if the index allows duplicate values. • SQL_FALSE if the index values must be unique. • NULL is returned if the TYPE column indicates that this row is SQL_TABLE_STAT (statistics information on the table itself).
5 INDEX_QUALIFIER	VARCHAR(128)	The string is used to qualify the index name in the DROP INDEX statement. Appending a period (.) plus the INDEX_NAME results in a full specification of the index.
6 INDEX_NAME	VARCHAR(128)	The name of the index. If the TYPE column has the value SQL_TABLE_STAT, this column has the value NULL.

SQLStatistics

Table 158. Columns returned by SQLStatistics (continued)

Column number/name	Data type	Description
7 TYPE	SMALLINT NOT NULL	Indicates the type of information contained in this row of the result set: <ul style="list-style-type: none"> • SQL_TABLE_STAT - Indicates this row contains statistics information on the table itself. • SQL_INDEX_CLUSTERED - Indicates this row contains information on an index, and the index type is a clustered index. • SQL_INDEX_HASHED - Indicates this row contains information on an index, and the index type is a hashed index. • SQL_INDEX_OTHER - Indicates this row contains information on an index, and the index type is other than clustered or hashed.
8 ORDINAL_POSITION	SMALLINT	Ordinal position of the column within the index whose name is given in the INDEX_NAME column. A NULL value is returned for this column if the TYPE column has the value of SQL_TABLE_STAT.
9 COLUMN_NAME	VARCHAR(128)	Name of the column in the index. A NULL value is returned for this column if the TYPE column has the value of SQL_TABLE_STAT.
10 ASC_OR_DESC	CHAR(1)	Sort sequence for the column; A for ascending, D for descending. NULL value is returned if the value in the TYPE column is SQL_TABLE_STAT.
11 CARDINALITY	INTEGER	<ul style="list-style-type: none"> • If the TYPE column contains the value SQL_TABLE_STAT, this column contains the number of rows in the table. • If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of unique values in the index. • A NULL value is returned if information is not available from the DBMS.
12 PAGES	INTEGER	<ul style="list-style-type: none"> • If the TYPE column contains the value SQL_TABLE_STAT, this column contains the number of pages used to store the table. • If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of pages used to store the indexes. • A NULL value is returned if information is not available from the DBMS.
13 FILTER_CONDITION	VARCHAR(128)	If the index is a filtered index, this is the filter condition. Since DATABASE 2 servers do not support filtered indexes, NULL is always returned. NULL is also returned if TYPE is SQL_TABLE_STAT.

For the row in the result set that contains table statistics (TYPE is set to SQL_TABLE_STAT), the columns values of NON_UNIQUE, INDEX_QUALIFIER, INDEX_NAME, ORDINAL_POSITION, COLUMN_NAME, and ASC_OR_DESC are set to NULL. If the CARDINALITY or PAGES information cannot be determined, then NULL is returned for those columns.

Note: The accuracy of the information returned in the SQLERRD(3) and SQLERRD(4) fields is dependent on many factors such as the use of parameter markers and expressions within the statement. The main factor

which can be controlled is the accuracy of the database statistics. That is, when the statistics were last updated, (for example, for DB2 for OS/390 and z/OS, the last time the RUNSTATS utility was run.)

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 159. SQLStatistics SQLSTATES

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS. The valid of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the SQLGetInfo() function.
HY100	Uniqueness option type out of range.	An invalid <i>fUnique</i> value was specified.
HY101	Accuracy option type out of range.	An invalid <i>fAccuracy</i> value was specified.
HYC00	Driver not capable.	DB2 ODBC does not support <i>catalog</i> as a qualifier for table name.

Restrictions

None.

Example

```
/* ... */
SQLRETURN
list_stats(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename)
{
/* ... */
    rc = SQLStatistics(hstmt, NULL, 0, schema, SQL_NTS,
                      tablename, SQL_NTS, SQL_INDEX_UNIQUE, SQL_QUICK);
    rc = SQLBindCol(hstmt, 4, SQL_C_SHORT,
                   &non_unique, 2, &non_unique_ind);
    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR,
                   index_name.s, 129, &index_name.ind);
    rc = SQLBindCol(hstmt, 7, SQL_C_SHORT,
                   &type, 2, &type_ind);
    rc = SQLBindCol(hstmt, 9, SQL_C_CHAR,
                   column_name.s, 129, &column_name.ind);
    rc = SQLBindCol(hstmt, 11, SQL_C_LONG,
                   &cardinality, 4, &card_ind);
    rc = SQLBindCol(hstmt, 12, SQL_C_LONG,
                   &pages, 4, &pages_ind);

    printf("Statistics for %s.%s\n", schema, tablename);

    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
    { if (type != SQL_TABLE_STAT)
      { printf(" Column: %-18s Index Name: %-18s\n",
              column_name.s, index_name.s);
        }
      else
      { printf(" Table Statistics:\n");
        }
      if (card_ind != SQL_NULL_DATA)
        printf(" Cardinality = %13ld", cardinality);
      else
        printf(" Cardinality = (Unavailable)");

      if (pages_ind != SQL_NULL_DATA)
        printf(" Pages = %13ld\n", pages);
      else
        printf(" Pages = (Unavailable)\n");
    }
/* ... */
}
```

References

- “SQLColumns - Get column information for a table” on page 124
- “SQLSpecialColumns - Get special (row identifier) columns” on page 369

SQLTablePrivileges - Get privileges associated with a table

Purpose

Specification:	ODBC 1.0		
-----------------------	-----------------	--	--

SQLTablePrivileges() returns a list of tables and associated privileges for each table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

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

Function arguments

Table 160. SQLTablePrivileges arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLCHAR *	szTableQualifier	Input	Catalog qualifier of a 3 part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	cbTableQualifier	Input	Length of <i>szCatalogName</i> . This must be set to 0.
SQLCHAR *	szSchemaName	Input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by schema name.
SQLSMALLINT	cbSchemaName	Input	Length of <i>szSchemaName</i> .
SQLCHAR *	szTableName	Input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by table name.
SQLSMALLINT	cbTableName	Input	Length of <i>szTableName</i> .

The *szSchemaName* and *szTableName* arguments accept search pattern. For more information about valid search patterns, see “Input arguments on catalog functions” on page 398.

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.

Since calls to SQLTablePrivileges() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128

SQLTablePrivileges

characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the `SQL_MAX_CATALOG_NAME_LEN`, `SQL_MAX_OWNER_SCHEMA_LEN`, `SQL_MAX_TABLE_NAME_LEN`, and `SQL_MAX_COLUMN_NAME_LEN` to determine respectively the actual lengths of the `TABLE_CAT`, `TABLE_SCHEM`, `TABLE_NAME`, and `COLUMN_NAME` columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 161. Columns returned by `SQLTablePrivileges`

Column Number/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: <ul style="list-style-type: none">• ALTER• CONTROL• DELETE• INDEX• INSERT• REFERENCES• SELECT• UPDATE
7 IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users. This can be "YES", "NO" or NULL.

Note: 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

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 162. `SQLTablePrivileges` `SQLSTATES`

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (<code>SQLParamData()</code> , <code>SQLPutData()</code>) operation.

Table 162. SQLTablePrivileges SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal to SQL_NTS. The value of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the SQLGetInfo() function.
HYC00	Driver not capable.	DB2 ODBC does not support <i>catalog</i> as a qualifier for table name.

Restrictions

None.

Example

```

/* ... */
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 */
    /* ... */

```

References

- “SQLTables - Get table information” on page 382

SQLTables - Get table information

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	
-----------------------	-----------------	-------------------	--

SQLTables() returns a list of table names and associated information stored in the system catalog of the connected data source. The list of table names is returned as a result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN SQLTables(
    (SQLHSTMT hstmt,
     SQLCHAR FAR *szCatalogName,
     SQLSMALLINT cbCatalogName,
     SQLCHAR FAR *szSchemaName,
     SQLSMALLINT cbSchemaName,
     SQLCHAR FAR *szTableName,
     SQLSMALLINT cbTableName,
     SQLCHAR FAR *szTableType,
     SQLSMALLINT cbTableType);
```

Function arguments

Table 163. SQLTables arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLCHAR *	szCatalogName	Input	Buffer that can contain a <i>pattern-value</i> to qualify the result set. <i>Catalog</i> is the first part of a 3 part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbCatalogName	Input	Length of <i>szCatalogName</i> . This must be set to 0.
SQLCHAR *	szSchemaName	Input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by schema name.
SQLSMALLINT	cbSchemaName	Input	Length of <i>szSchemaName</i> .
SQLCHAR *	szTableName	Input	Buffer that can contain a <i>pattern-value</i> to qualify the result set by table name.
SQLSMALLINT	cbTableName	Input	Length of <i>szTableName</i> .
SQLCHAR *	szTableType	Input	Buffer that can contain a <i>value list</i> to qualify the result set by table type. The value list is a list of upper-case comma-separated single quoted values for the table types of interest. Valid table type identifiers can include: TABLE, VIEW, SYSTEM TABLE, ALIAS, SYNONYM. If <i>szTableType</i> argument is a NULL pointer or a zero length string, then this is equivalent to specifying all of the possibilities for the table type identifier. If SYSTEM TABLE is specified, then both system tables and system views (if there are any) are returned.
SQLSMALLINT	cbTableType	Input	Size of <i>cbTableType</i>

Note that the *szCatalogName*, *szSchemaName*, and *szTableName* arguments accept search patterns. For more information about valid search patterns, see “Input arguments on catalog functions” on page 398.

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 NULLs.)

If *szTableType* is not an empty string, it must contain a list of upper-case, comma-separated values for the types of interest; each value can be enclosed in single quotes or unquoted. For example, `"'TABLE','VIEW'"` or `"TABLE,VIEW"`. If the data source does not support or does not recognize a specified table type, nothing is returned for that type.

Sometimes, an application calls `SQLTables()` with null pointers for some or all of the *szSchemaName*, *szTableName*, and *szTableType* arguments so that no attempt is made to restrict the result set returned. For some data sources that contain a large quantity of tables, views, or aliases, this scenario maps to an extremely large result set and very long retrieval times. Three mechanisms are introduced to help the end user reduce the long retrieval times: three keywords (`SCHEMALIST`, `SYSCHEMA`, `TABLETYPE`) can be specified in the DB2 ODBC initialization file to help restrict the result set when the application has supplied null pointers for either or both of *szSchemaName* and *szTableType*. These keywords and their usage are discussed in detail in “Initialization keywords” on page 55. If the application did not specify a null pointer for *szSchemaName* or *szTableType* then the associated keyword specification in the DB2 ODBC initialization file is ignored.

The result set returned by `SQLTables()` contains the columns listed in Table 164 on page 384 in the order given. The rows are ordered by `TABLE_TYPE`, `TABLE_CAT`, `TABLE_SCHEM`, and `TABLE_NAME`.

Since calls to `SQLTables()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The `VARCHAR` columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the `SQL_MAX_CATALOG_NAME_LEN`, `SQL_MAX_OWNER_SCHEMA_LEN`, `SQL_MAX_TABLE_NAME_LEN`, and

SQLTables

SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 164. Columns returned by SQLTables

Column Name	Data type	Description
TABLE_CAT	VARCHAR(128)	The name of the catalog containing TABLE_SCHEM. This column contains a NULL value.
TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
TABLE_NAME	VARCHAR(128)	The name of the table, or view, or alias, or synonym.
TABLE_TYPE	VARCHAR(128)	Identifies the type given by the name in the TABLE_NAME column. It can have the string values 'TABLE', 'VIEW', 'INOPERATIVE VIEW', 'SYSTEM TABLE', 'ALIAS', or 'SYNONYM'.
REMARKS	VARCHAR(254)	Contains the descriptive information about the table.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 165. SQLTables SQLSTATES

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 08S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
HY001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
HY010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY014	No more handles.	DB2 ODBC is not able to allocate a handle due to internal resources.
HY090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal to SQL_NTS. The value of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the SQLGetInfo() function.
HYC00	Driver not capable.	DB2 ODBC does not support <i>catalog</i> as a qualifier for table name.

Restrictions

None.

Example

Also, see “Querying environment information example” on page 38.

```

/* ... */
SQLRETURN init_tables(SQLHDBC hdbc )
{
    SQLHSTMT      hstmt;
    SQLRETURN     rc;

    SQLUSMALLINT  rowstat[MAX_TABLES];
    SQLUIINTEGER  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_1);

    rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) &table->name, 129,
        &table->name_1);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) &table->type, 129,
        &table->type_1);

    rc = SQLBindCol(hstmt, 5, SQL_C_CHAR, (SQLPOINTER) &table->remarks, 255,
        &table->remarks_1);

    /* Now fetch the result set */
    /* ... */

```

References

- “SQLColumns - Get column information for a table” on page 124
- “SQLTablePrivileges - Get privileges associated with a table” on page 379

SQLTransact - Transaction management

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI
-----------------------	-----------------	-------------------	----------------

In ODBC 3.0, `SQLEndTran()` replaces the ODBC 2.0 function `SQLTransact()`. See `SQLEndTran()` for more information.

`SQLTransact()` commits or rolls back the current transaction in the specified connection. `SQLTransact()` can also be used to request that a commit or rollback be issued for each of the connections associated with the environment.

All changes to the database performed on the connection since connect time or the previous call to `SQLTransact()` (whichever is the most recent) are committed or rolled back.

If a transaction is active on a connection, the application must call `SQLTransact()` before it can disconnect from the database.

Syntax

```
SQLRETURN SQLTransact(
    (SQLHENV   henv,
     SQLHDBC   hdbc,
     SQLUSMALLINT fType);
```

Function arguments

Table 166. *SQLTransact arguments*

Data type	Argument	Use	Description
SQLHENV	<i>henv</i>	input	Environment handle. If <i>hdbc</i> is a valid connection handle, <i>henv</i> is ignored.
SQLHDBC	<i>hdbc</i>	input	Database connection handle. If <i>hdbc</i> is set to <code>SQL_NULL_HDBC</code> , then <i>henv</i> must contain the environment handle that the connection is associated with.
SQLUSMALLINT	<i>fType</i>	input	The desired action for the transaction. The value for this argument must be one of: <ul style="list-style-type: none"> SQL_COMMIT SQL_ROLLBACK

Usage

In DB2 ODBC, a transaction begins implicitly when an application that does not already have an active transaction, issues `SQLPrepare()`, `SQLExecDirect()`, `SQLExecDirect()`, `SQLGetTypeInfo()`, or one of the catalog functions. The transaction ends when the application calls `SQLTransact()` or disconnects from the data source.

If the input connection handle is `SQL_NULL_HDBC` and the environment handle is valid, then a commit or rollback is issued on each of the open connections in the environment. `SQL_SUCCESS` is returned only if success is reported on all the connections. If the commit or rollback fails for one or more of the connections,

SQLTransact() returns SQL_ERROR. To determine which connections failed the commit or rollback operation, the application needs to call SQLError() on each connection handle in the environment.

It is important to note that unless the connection option SQL_CONNECTTYPE is set to SQL_COORDINATED_TRANS (to indicate coordinated distributed transactions), there is no attempt to provide coordinated global transaction with one-phase or two-phase commit protocols.

Completing a transaction has the following effects:

- Prepared SQL statements (using SQLPrepare()) survive transactions; they can be executed again without first calling SQLPrepare().
- Cursor positions are maintained after a commit unless one or more of the following is true:
 - The server is DB2 Server for VSE & VM.
 - The SQL_CURSOR_HOLD statement option 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 has not been overridden by resetting the SQL_CURSOR_HOLD statement option.
 - The CURSORHOLD keyword is present in a the connection string on the SQLDriverConnect() call that set up this connection, and it indicates cursor with hold is not in effect, and this has not been overridden by resetting the SQL_CURSOR_HOLD statement option.

If the cursor position is not maintained due to any one of the above circumstances, the cursor is closed and all pending results are discarded.

If the cursor position is maintained after a commit, the application must issue a fetch to re-position the cursor (to the next row) before continuing with processing of the remaining result set.

To determine whether cursor position is maintained after a commit, call SQLGetInfo() with the SQL_CURSOR_COMMIT_BEHAVIOR information type.

- Cursors are closed after a rollback and all pending results are discarded.
- Statement handles are still valid after a call to SQLTransact(), and can be reused for subsequent SQL statements or de-allocated by calling SQLFreeStmt().
- Cursor names, bound parameters, and column bindings survive transactions.

If no transaction is currently active on the connection, calling SQLTransact() has no effect on the database server and returns SQL_SUCCESS.

SQLTransact() can fail while executing the COMMIT or ROLLBACK due to a loss of connection. In this case the application might not be able to determine whether the COMMIT or ROLLBACK was processed, and a database administrator's help might be required. See the DBMS product information for more information on transaction logs and other transaction management tasks.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

SQLTransact

Diagnostics

Table 167. SQLTransact SQLSTATES

SQLSTATE	Description	Explanation
08003	Connection is closed.	The <i>hdbc</i> is not in a connected state.
08007	Connection failure during transaction.	The connection associated with the <i>hdbc</i> failed during the execution of the function and it cannot be determined whether the requested COMMIT or ROLLBACK occurred before the failure.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 ODBC is not able to allocate memory required to support execution or completion of the function.
S1012	Invalid transaction code.	The value specified for the argument <i>fType</i> was neither SQL_COMMIT not SQL_ROLLBACK.
S1013	Unexpected memory handling error.	DB2 ODBC is not able to access memory required to support execution or completion of the function.

Restrictions

SQLTransact() can not be issued if the application is executing as a stored procedure.

Example

See “Example” on page 180.

References

- “SQLSetStmtOption - Set statement option” on page 367
- “SQLGetInfo - Get general information” on page 234

Chapter 6. Using advanced features

This section covers a series of advanced tasks.

- “Environment, connection, and statement options”
- “Distributed unit of work (coordinated distributed transactions)” on page 391
- “Global transaction processing” on page 396
- “Querying system catalog information” on page 397
- “Sending/retrieving long data in pieces” on page 401
- “Using arrays to input parameter values” on page 403
- “Retrieving a result set into an array” on page 406
- “Using large objects” on page 411
- “Using distinct types” on page 414
- “Using stored procedures” on page 417
- “Writing multithreaded applications” on page 421
- “Using Unicode functions” on page 429
- “Mixing embedded SQL and DB2 ODBC” on page 446
- “Using vendor escape clauses” on page 448
- “Programming hints and tips” on page 452

Environment, connection, and statement options

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

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

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

- Some can be set any time after the connection handle is allocated.
- Some can be set only before the actual connection is established.
- Some can be set only after the connection is established.
- Some can be set after the connection is established, but only while there are no outstanding transactions or open cursors.

The application can change the value of connection options by calling `SQLSetConnectAttr()` and can obtain the current value of an option by calling `SQLGetConnectAttr()`. An example of a connection option which can be set any time after a handle is allocated is the auto-commit option introduced in “Commit or rollback” on page 26. For complete details on when each option can be set, see “`SQLSetConnectOption` - Set connection option” on page 345.

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

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

The application can specify the value of any settable statement option by calling `SQLSetStmtAttr()`, and can obtain the current value of an option by calling `SQLGetStmtAttr()`. For complete details on when each option can be set, see “`SQLSetStmtOption - Set statement option`” on page 367.

The `SQLSetConnectAttr()` function can also be used to set statement options for all statement handles currently associated with the connection as well as for all future statement handles to be allocated under this connection. However, `SQLGetConnectAttr()` can only be used to obtain connection options; `SQLGetStmtAttr()` must be used to obtain the current value of a statement option.

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

The DB2 ODBC initialization file can be used to change default values for all DB2 ODBC applications. This might be the end user’s only means of changing the defaults if the application does not have a way for the user to provide default attribute values in the `SQLDriverConnect()` connection string. Default attribute values that are specified on `SQLDriverConnect()` override the values in the DB2 ODBC initialization file for that particular connection. For information on how the end user can use the DB2 ODBC initialization file as well as for a list of changeable defaults, see “DB2 ODBC initialization file” on page 52.

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

The options that can be changed, are listed in the detailed function descriptions of the *set* option or attributes functions, see Chapter 5, “Functions”, on page 67. The read-only options (if any exist) are listed with the detailed function descriptions of the *get* option or attribute functions.

For information on some commonly used options, see “Programming hints and tips” on page 452.

Figure 8 on page 391 shows the addition of the option or attribute functions to the basic connect scenario.

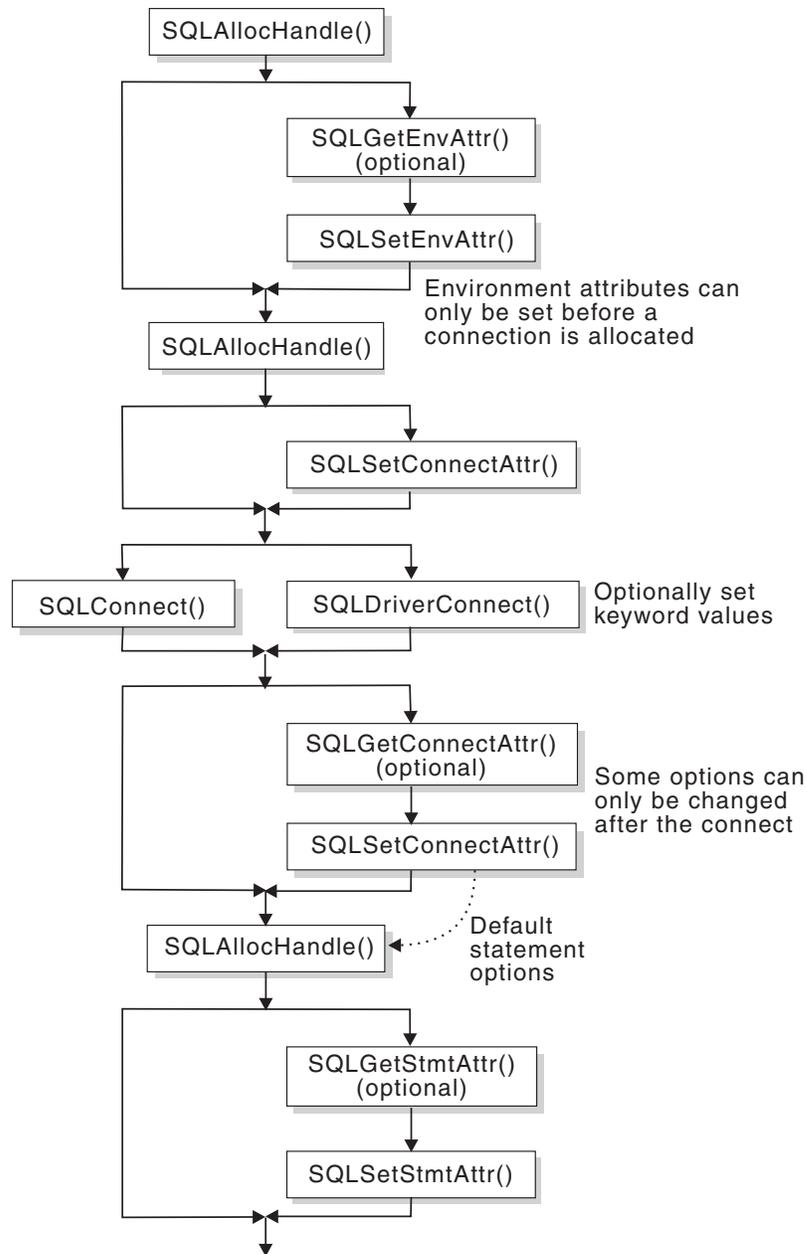


Figure 8. Setting and retrieving options (attributes)

Distributed unit of work (coordinated distributed transactions)

The transaction scenario described in “Connecting to one or more data sources” on page 18 portrays an application which interacts with only one database server in a transaction. Further, only one transaction (that associated with the current server) existed at any given time.

With distributed unit of work (coordinated distributed transactions), the application, if executing `CONNECT` (type 2), is able to access multiple database servers from within the same coordinated transaction. This section describes how DB2 ODBC applications can be written to use coordinated distributed unit of work.

First, consider the environment attribute (SQL_CONNECTTYPE) which controls whether the application is to operate in a coordinated or uncoordinated distributed environment. The two possible values for this attribute are:

- SQL_CONCURRENT_TRANS - supports the single data source per transaction semantics described in Chapter 2. Multiple (logical) concurrent connections to different data sources are permitted. This is the default.
- SQL_COORDINATED_TRANS - supports the multiple data sources per transaction semantics, as discussed below.

All connections within an application must have the same SQL_CONNECTTYPE setting.

Recommendation: Have the application set this environment attribute, if necessary, as soon as `SQLAllocHandle()` (with *HandleType* set to `SQL_HANDLE_ENV`) is called successfully.

Options that govern distributed unit of work semantics

A coordinated transaction means that commits or rollbacks among multiple data source connections are coordinated. The `SQL_COORDINATED_TRANS` setting of the `SQL_CONNECTTYPE` attribute corresponds to the `CONNECT` (type 2) in IBM embedded SQL.

All the connections within an application must have the same `SQL_CONNECTTYPE` setting. After the first connection is established, all subsequent connect types must be the same as the first. Coordinated connections default to manual-commit mode (for discussion on auto-commit mode, see “Commit or rollback” on page 26).

Figure 9 on page 393 shows the logical flow of an application executing statements on two `SQL_CONCURRENT_TRANS` connections ('A' and 'B'), and indicates the scope of the transactions.

Figure 10 on page 394 shows the same statements being executed on two `SQL_COORDINATED_TRANS` connections (A and B), and the scope of a coordinated distributed transaction.

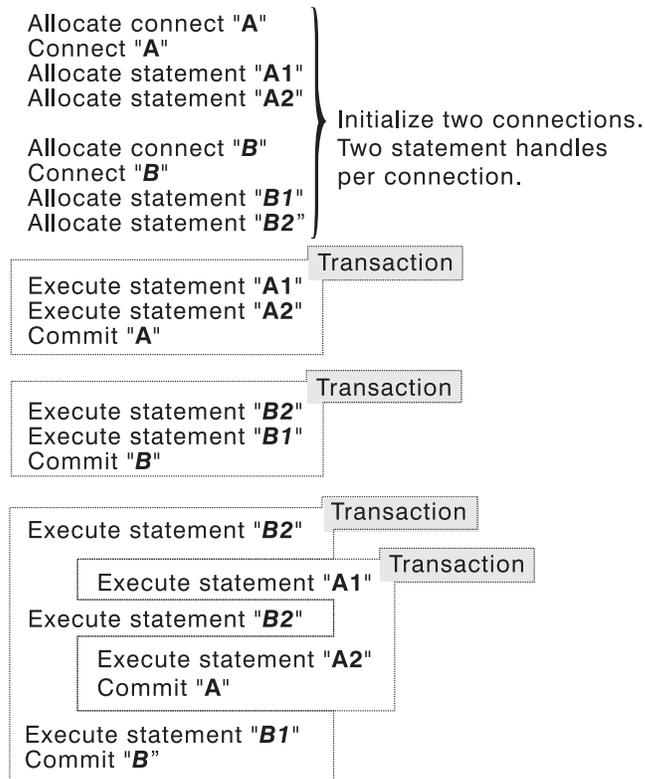


Figure 9. Multiple connections with concurrent transactions

In Figure 9, within the context of the ODBC connection model, the third and fourth transactions can be interleaved as shown. That is, if the application has specified `SQL_CONCURRENT_TRANS`, then the ODBC model supports one transaction for each active connection. The third transaction, consisting of the execution of statements B2, B2 again and B1 at data source B, can be managed and committed independent of the fourth transaction, consisting of the execution of statements A1 and A2 at data source A. That is, the transactions at A and B are independent and exist concurrently.

If the application specifies `SQL_CONCURRENT_TRANS` and is executing with `MULTICONTEXT=0` specified in the initialization file, then DB2 for OS/390 and z/OS allows any number of concurrent connection handles to be allocated, subject to the restriction that only one physical connection can exist at any given time. This behavior precludes support for the ODBC connection model, and consequently the behavior of the application is substantially different (than that described for the ODBC execution model described above.)

In particular, the third transaction is executed as three transactions. Prior to executing statement 'B2'" DB2 ODBC connects to B. This statement is executed and committed prior to reconnecting to data source A to execute A1. Similarly, this statement at data source A is committed prior to reconnecting to data source B to execute statement B2. This statement is then committed and a reconnection is made to A to execute A2. Next, another commit occurs and a reconnection to B to execute B1.

From an application point of view, the transaction at data source B, consisting of B2->B2->B1, is broken into three independent transactions: B2, B2 and B1. The

fourth transaction at data source A, consisting of A1->A2, is broken into two independent transactions: A1 and A2.

```

Allocate environment
Set environment attributes
(SQL_CONNECTTYPE)

Allocate connect "A"
Connect "A"
(SQL_CONCURRENT_TRANS)

Allocate statement "A1"
Allocate statement "A2"

Allocate connect "B"
Connect "B"
(SQL_CONCURRENT_TRANS)

Allocate statement "B1"
Allocate statement "B2"

```

} Initialize two connections.
Two statement handles
per connection.

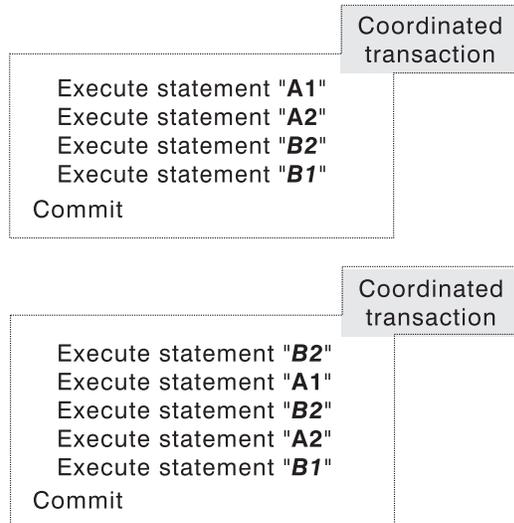


Figure 10. Multiple connections with coordinated transactions

For a discussion of multiple active transaction support, see “DB2 ODBC support of multiple contexts” on page 424.

Establishing a coordinated transaction connection

An application can establish coordinated transaction connections by calling the `SQLSetEnvAttr()` or `SQLSetConnectAttr()` functions, or by setting the `CONNECTTYPE` keyword in the DB2 ODBC initialization file or in the connection string for `SQLDriverConnect()`. The initialization file is intended for existing applications that do not use the `SQLSetConnectAttr()` function. For information about the keywords, see “DB2 ODBC initialization file” on page 52.

An application cannot have a mixture of concurrent and coordinated connections; the type of the first connection determines the type of all subsequent connections. `SQLSetEnvAttr()` and `SQLSetConnectAttr()` return an error if an application attempts to change the connect type while there is an active connection. When the connection type is established, it persists until `SQLFreeHandle` is called (with `HandleType` set to `SQL_HANDLE_ENV`).

Distributed unit of work example

The following example connects to two data sources using a SQL_CONNECTTYPE set to SQL_COORDINATED_TRANS (CONNECT (type 2)).

```
/* ... */
#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 Options */
    /* 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, etc. */
/* 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;

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

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

```

Global transaction processing

A global transaction is a recoverable unit of work, or transaction, comprised of changes to a collection of resources. All resources that participate in the global transaction are guaranteed to be committed or rolled back as an atomic unit under the control of OS/390 Transaction Management and Recoverable Resource Manager Services (OS/390 RRS) using the two-phase commit protocol. A resource manager registered with RRS manages and controls access to each resource. When an application requests access to a resource, the resource manager must express interest in the transaction that the application is processing. The expressed interest informs RRS that the resource manager is involved with the transaction and, in response, RRS keeps the resource manager informed of transaction-related events.

When the application is ready to commit or back out its changes, RRS globally coordinates all changes within the transaction as they commit or roll back. When all resource processing is complete, global transaction processing is complete.

A DB2 ODBC application can participate in a global transaction by clearly demarcating the transaction boundaries. Marking the boundaries ensures that transactions on a global connection complete global transaction processing.

- To mark the beginning of a global transaction, an ODBC application must call the RRS service, ATRBEG.
- To mark the end of a global transaction, the application must call SRRCMIT and SRRBACK functions or the RRS service, ATREND.

Specify the keywords `AUTOCOMMIT=0`, `MULTICONTEXT=0`, and `MVSATTACHTYPE=RRSAF` in the initialization file to enable global transaction processing.

Global transaction support requires OS/390 Version 2 Release 8 or higher. DB2 ODBC does not support global transaction processing for applications that run under a stored procedure.

For a complete description of RRS callable services, see *OS/390 MVS Programming: Assembler Services Guide* or *OS/390 MVS Programming: Resource Recovery*.

Querying system catalog information

Often, one of the first tasks an application performs is to display a list of tables from which the user selects one or more tables to work with. Although the application can issue its own queries against the database system catalog to get this type of catalog information, it is best for the application to either call the DB2 ODBC catalog query functions or direct the catalog query functions to the DB2 ODBC shadow catalog.

The following sections describe the methods you can use to query the system catalog.

- “Using the catalog query functions”
- “Directing catalog queries to the DB2 ODBC shadow catalog” on page 399

Using the catalog query functions

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

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

- “SQLColumnPrivileges - Get privileges associated with the columns of a table” on page 120
- “SQLColumns - Get column information for a table” on page 124
- “SQLForeignKeys - Get the list of foreign key columns” on page 181
- “SQLPrimaryKeys - Get primary key columns of a table” on page 308
- “SQLProcedureColumns - Get input/output parameter information for a procedure” on page 313
- “SQLProcedures - Get list of procedure names” on page 323
- “SQLSpecialColumns - Get special (row identifier) columns” on page 369

- “SQLStatistics - Get index and statistics information for a base table” on page 374
- “SQLTablePrivileges - Get privileges associated with a table” on page 379
- “SQLTables - Get table information” on page 382

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

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

Input arguments on catalog functions

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

In the Function Arguments sections for these catalog functions in Chapter 5, “Functions”, on page 67, each of the above input arguments are described either as a pattern-value or just as an ordinary argument. For example, `SQLColumnPrivileges()` treats *SchemaName* and *TableName* as ordinary arguments and *ColumnName* as a pattern-value.

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

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

Each pattern-value argument can contain:

- The underscore (`_`) character which stands for any single character.
- The percent (`%`) character which stands for any sequence of zero or more characters.
- Characters which 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 the metadata characters (`_`, `%`) as themselves, an escape character must immediately precede the `_` or `%`. The escape character itself can be specified as part of the pattern by including it twice in succession. An application can determine the escape character by calling `SQLGetInfo()` with `SQL_SEARCH_PATTERN_ESCAPE`.

Catalog functions example

In the sample application:

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

DB2 UDB is the data source in this example. Output from a sample is shown below. Relevant segments of the sample are listed for each of the catalog functions.

```
Enter Search Pattern for Table Schema Name:
STUDENT
Enter Search Pattern for Table Name:
%
### TABLE SCHEMA          TABLE_NAME          TABLE_TYPE
-----
1  STUDENT                  CUSTOMER             TABLE
2  STUDENT                  DEPARTMENT          TABLE
3  STUDENT                  EMP_ACT             TABLE
4  STUDENT                  EMP_PHOTO           TABLE
5  STUDENT                  EMP_RESUME          TABLE
6  STUDENT                  EMPLOYEE            TABLE
7  STUDENT                  NAMEID              TABLE
8  STUDENT                  ORD_CUST            TABLE
9  STUDENT                  ORD_LINE            TABLE
10 STUDENT                  ORG                 TABLE
11 STUDENT                  PROD_PARTS          TABLE
12 STUDENT                  PRODUCT             TABLE
13 STUDENT                  PROJECT             TABLE
14 STUDENT                  STAFF               TABLE
Enter a table Number and an action:(n [Q | C | P | I | F | T | 0 | 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<<
```

Directing catalog queries to the DB2 ODBC shadow catalog

DB2 ODBC applications that frequently use ODBC catalog functions to access the DB2 system catalog can redirect catalog queries to the DB2 ODBC shadow catalog.

The shadow catalog consists of a set of pseudo-catalog tables that contain rows representing objects defined in the DB2 system catalog. These tables are pre-joined and indexed to provide faster catalog access for ODBC applications. The table rows are limited to the columns necessary for supporting ODBC operations.

The tables that comprise the DB2 ODBC shadow catalog have a default schema name of CLISCHEM. To redirect catalog functions to access the base DB2 ODBC shadow catalog tables, you must 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 under 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 to the data source section of the DB2 ODBC initialization file as follows:

```
[DATASOURCE]
MVSDEFAULTSSID=V61A
CLISCHEMA=PAYROLL
APPLTRACE=1
APPLTRACEFILENAME="DD:APPLTRC"
```

where `myschema` is the schema name of the set of views you create.

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. For example:

```
CREATE VIEW PAYROLL.table_name AS
  SELECT * FROM PAYROLL.table_name WHERE TABLE_SCHEM='USER01';
```

where `table_name` is the name of a DB2 ODBC shadow catalog table (for example, COLUMNPRIVILEGES).

Table 168 lists the DB2 ODBC shadow catalog tables and the catalog functions used to access the tables.

Table 168. Shadow catalog tables and DB2 ODBC APIs

Shadow catalog table	DB2 ODBC catalog function
CLISCHEM.COLUMNPRIVILEGES	SQLColumnPrivileges()
CLISCHEM.COLUMNS	SQLColumns()
CLISCHEM.FOREIGNKEYS	SQLForeignKeys()
CLISCHEM.PRIMARYKEYS	SQLPrimaryKeys()
CLISCHEM.PROCEDURECOLUMNS	SQLProcedureColumns()
CLISCHEM.PROCEDURES	SQLProcedures()
CLISCHEM.SPECIALCOLUMNS	SQLSpecialColumns()
CLISCHEM.TSTATISTICS	SQLStatistics()
CLISCHEM.TABLEPRIVILEGES	SQLTablePrivileges()
CLISCHEM.TABLE	SQLTables()

Creating and maintaining the DB2 ODBC shadow catalog

DB2 DataPropagator for OS/390 populates and maintains the DB2 ODBC shadow catalog. DB2 for OS/390 and z/OS supports the DATA CAPTURE CHANGE clause of the ALTER TABLE SQL statement for DB2 system catalog tables. This support allows DB2 to mark log records associated with any statements that change the DB2 catalog (for example, CREATE and DROP). In addition, the DB2 DataPropagator Capture and Apply process identifies and propagates the DB2 system catalog changes to the DB2 ODBC shadow based on marked log records.

For detailed information about setting up the DB2 ODBC shadow catalog and running the DB2 DataPropagator Capture and Apply programs, see *DB2 UDB Replication Guide and Reference*.

Shadow catalog example

If you specify CLISHEMA=PAYROLL in the data source section of the DB2 ODBC initialization file, the ODBC catalog functions that normally query the DB2 system catalog tables (SYSIBM schema) will reference the following set of views of the ODBC shadow catalog base tables:

- PAYROLL.COLUMNS
- PAYROLL.TABLES
- PAYROLL.COLUMNPRIVILIGES
- PAYROLL.TABLEPRIVILIGES
- PAYROLL.SPECIALCOLUMNS
- PAYROLL.PRIMARYKEYS
- PAYROLL.FOREIGNKEYS
- PAYROLL.TSTATISTICS
- PAYROLL.PROCEDURES
- PAYROLL.PROCEDURECOLUMNS

Sending/retrieving long data in pieces

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

Specifying parameter values at execute time

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

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

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

1. It calls `SQLParamData()` to conceptually advance to the first such parameter. `SQLParamData()` returns `SQL_NEED_DATA` and provides the contents of the

input data pointer argument specified on the associated `SQLBindParameter()` or `SQLSetParam()` call to help identify the information required.

2. It calls `SQLPutData()` to pass the actual data for the parameter. Long data can be sent in pieces by calling `SQLPutData()` repeatedly.
3. It calls `SQLParamData()` again after it has provided the entire data for this data-at-execute parameter. If more data-at-execute parameters exist, `SQLParamData()` again returns `SQL_NEED_DATA` and the application repeats steps 2 and 3 above.

When all data-at-execute parameters are assigned values, `SQLParamData()` completes execution of the SQL statement and produces a return value and diagnostics as the original `SQLExecDirect()` or `SQLExecute()` would have produced. The right side of Figure 11 on page 403 illustrates this flow.

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

- `SQLParamData()` and `SQLPutData()` as given in the sequence above.
- The `SQLCancel()` function which is used to cancel the flow and force an exit from the loops on the right side of Figure 11 on page 403 without executing the SQL statement.
- The `SQLGetDiagRec()` function. The application also must not end the transaction nor set any connection attributes.

Fetching data in pieces

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

Basically, as shown on the left side of Figure 11 on page 403, a call to `SQLGetData()` returns `SQL_SUCCESS_WITH_INFO` (with `SQLSTATE 01004`) to indicate more data exists for this column. `SQLGetData()` is called repeatedly to get the remaining pieces of data until it returns `SQL_SUCCESS`, signifying that the entire data were retrieved for this column.

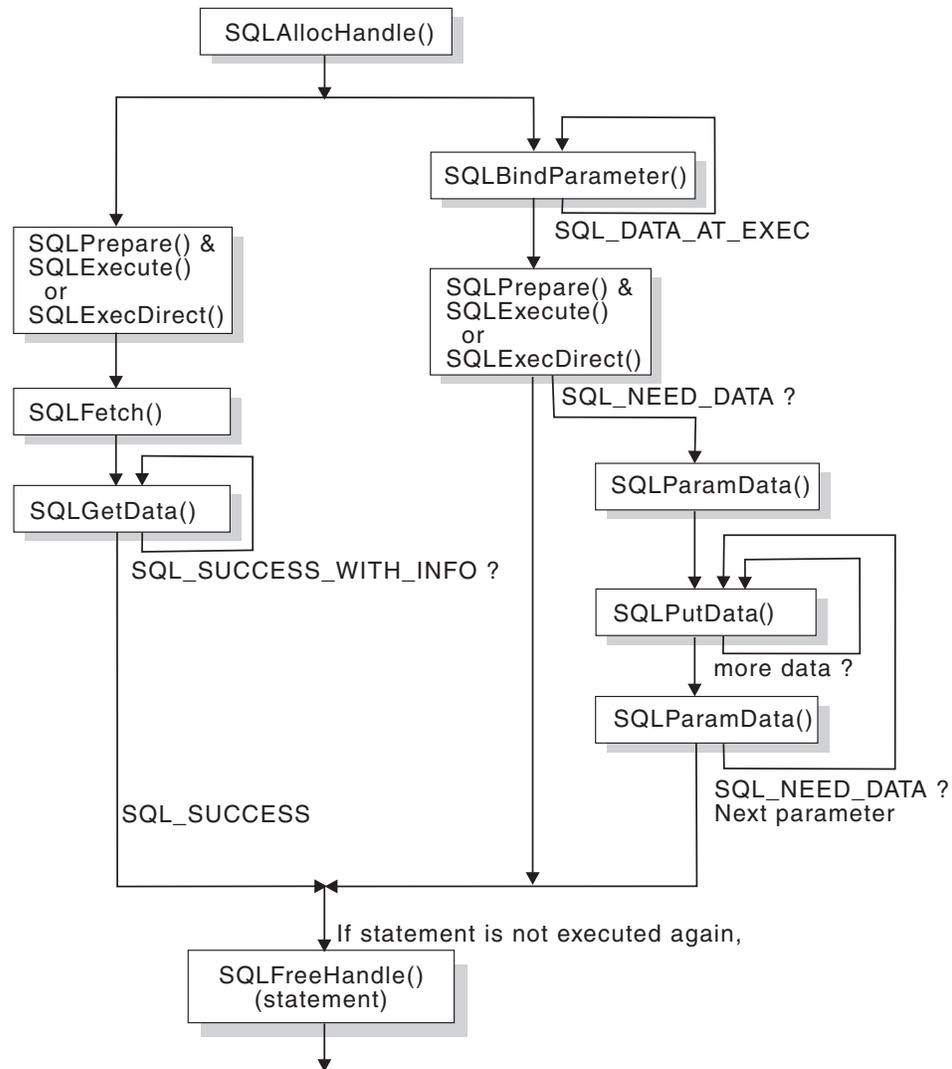


Figure 11. Input and retrieval of data in pieces

Using arrays to input parameter values

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

This method involves the binding of parameter markers to arrays of storage locations using the `SQLBindParameter()` call. For character and binary input data, the application uses the maximum input buffer size argument (`cbValueMax`) on `SQLBindParameter()` call to indicate to DB2 ODBC the location of values in the input array. For other input data types, the length of each element in the array is assumed to be the size of the C data type. `SQLParamOptions()` is called to specify how many elements are in the array before the execution of the SQL statement.

Suppose for Figure 12 on page 405 there is an application that allows the user to change values in the OVERTIME_WORKED and OVERTIME_PAID columns of a time sheet data entry form. Also suppose that the primary key of the underlying EMPLOYEE table is EMPLOY_ID. The application can then request to prepare the following SQL statement:

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

When the user enters all the changes, the application counts that n rows are to change and allocates $m=3$ arrays to store the changed data and the primary key. Then, the application makes calls to functions as follows:

1. SQLBindParameter() to bind the three parameter markers to the location of three arrays in memory.
2. SQLParamOptions() to specify the number of rows to change (the size of the array).
3. SQLExecute() once and all the updates are sent to the database.

This is the flow shown on the right side of Figure 12 on page 405.

The basic method is shown on the left side of Figure 12 on page 405 where SQLBindParameter() is called to bind the three parameter markers to the location of three variables in memory. SQLExecute() is called to send the first set of changes to the database. The variables are updated to reflect values for the next row of changes and again SQLExecute() is called. This method has $n-1$ extra SQLExecute() calls. For insert and update, use SQLRowCount to verify the number of changed rows.

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

For queries with parameter markers on the WHERE clauses, an array of input values generate multiple sequential result sets. Each result set can be processed before moving onto the next one by calling SQLMoreResults(). See "SQLMoreResults - Determine if there are more result sets" on page 286 for more information and an example.

Figure 12 on page 405 shows the two methods of executing a statement with m parameters n times. The basic method shown on the left calls SQLExecute() once for each set of parameter values. The array method on the right, calls SQLParamOptions() to specify the number of rows (n), then calls SQLExecute() once. Both methods must call SQLBindParameter() once for each parameter.

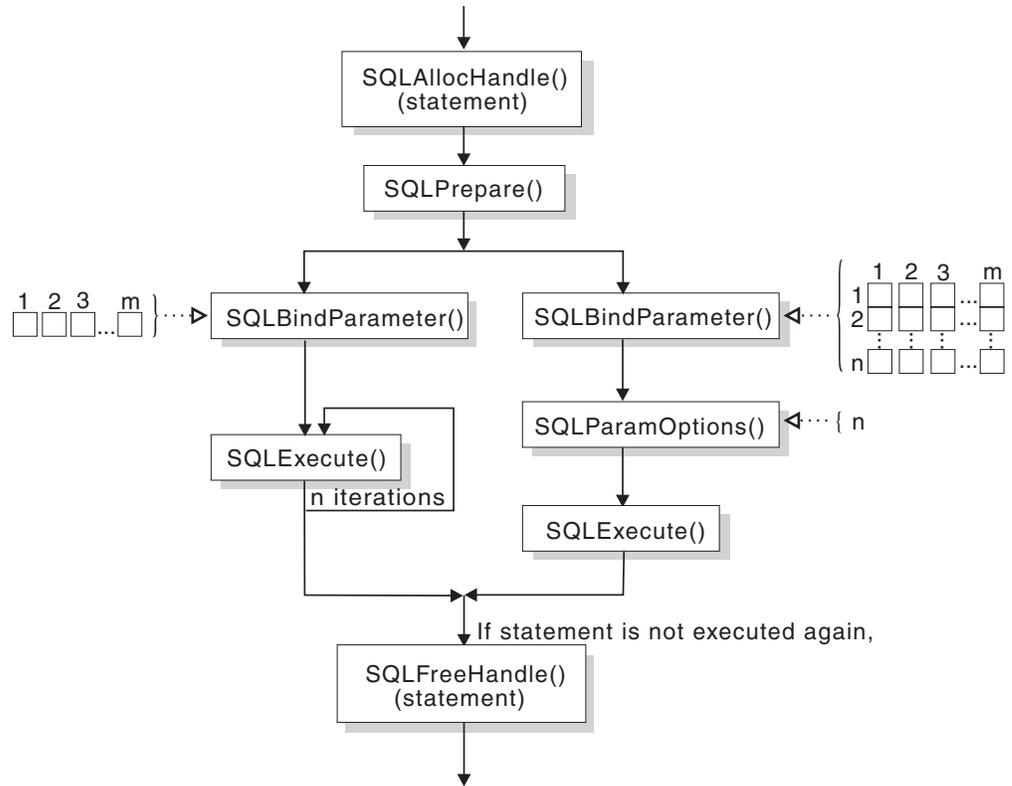


Figure 12. Array insert

Array input example

This example shows an array INSERT statement. For an example of an array query statement, see “SQLMoreResults - Determine if there are more result sets” on page 286.

```

/* ... */
SQLINTEGER 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", "LUCCHESE", "O'CONNELL", "QUINTANA",
    "NICHOLLS", "ADAMSON", "PIANKA", "YOSHIMURA",
    "SCOUTTEN", "WALKER", "BROWN", "JONES",
    "LUTZ", "JEFFERSON", "MARINO", "SMITH",
    "JOHNSON", "PEREZ", "SCHNEIDER", "PARKER",
    "SMITH", "SETRIGHT", "MEHTA", "LEE",
    "GOUNOT"
};
/* ... */
/* Prepare the statement */
rc = SQLPrepare(hstmt, stmt, SQL_NTS);

rc = SQLParamOptions(hstmt, 25, &pirow);

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

rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
    31, 0, First_Name, 31, NULL);

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

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

/* ... */

```

Retrieving a result set into an array

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

Alternatively, applications can eliminate the overhead of extra data copies or extra `SQLBindCol()` calls by retrieving multiple rows of data (called a rowset) at a time into an array. `SQLBindCol()` is also used to assign storage for application array variables. By default, the binding of rows is in column-wise fashion: this is

symmetrical to using `SQLBindParameter()` to bind arrays of input parameter values as described in the previous section.

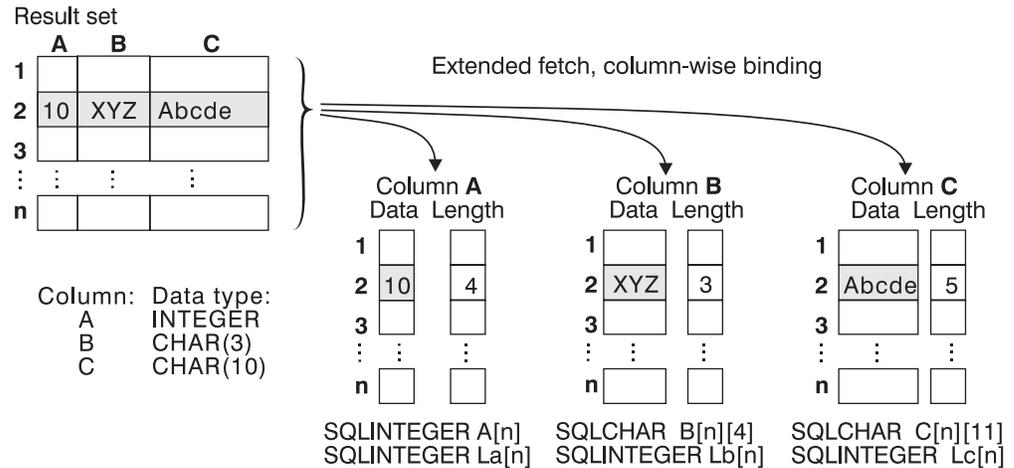


Figure 13. Column-wise binding

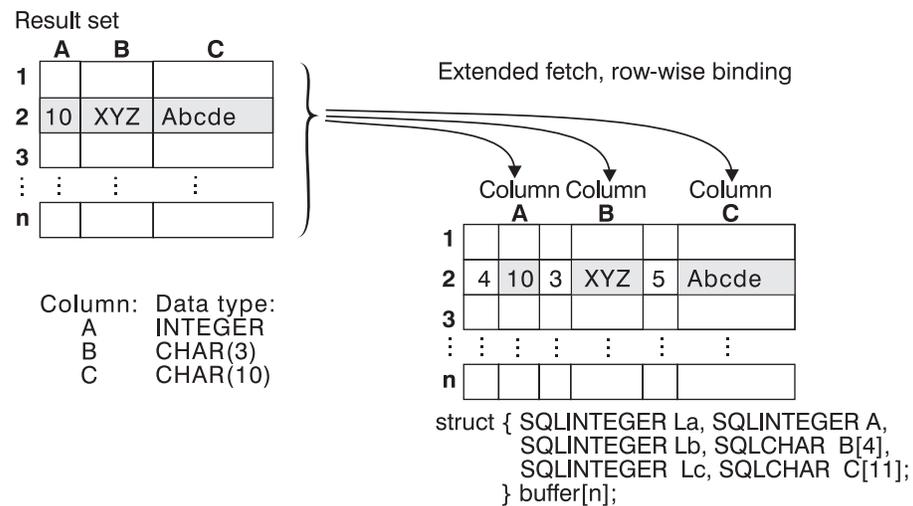


Figure 14. Row-wise binding

Returning array data for column-wise bound data

Figure 13 is a logical view of column-wise binding. The right side of Figure 15 on page 409 shows the function flows for column-wise retrieval.

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

The application then calls `SQLExtendedFetch()` to retrieve the data. When returning data, DB2 ODBC uses the maximum buffer size argument (`cbValueMax`) on `SQLBindCol()` to determine where to store successive rows of data in the array; the number of bytes available for return for each element is stored in the deferred

length array. If the number of rows in the result set is greater than the `SQL_ROWSET_SIZE` attribute value, multiple calls to `SQLExtendedFetch()` are required to retrieve all the rows.

Returning array data for row-wise bound data

The application can also do row-wise binding which associates an entire row of the result set with a structure. In this case the rowset is retrieved into an array of structures, each of which holds the data in one row and the associated length fields. Figure 14 on page 407 gives a pictorial view of row-wise binding.

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

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

Figure 15 on page 409 shows the required functions for each method. The left side shows n rows being selected, and retrieved one row at a time into m application variables. The right side shows the same n rows being selected, and retrieved directly into an array.

Note:

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


```

        /* 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)/ROW_SIZE, 0);

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

    /* Bind remaining columns ... */
    /* ... */

```

```

/* NOTE: This sample assumes that an order never has more
        rows than ROWSET_SIZE. A check should be added below to call
        SQLExtendedFetch multiple times for each result set.
*/
do /* for each result set .... */
{ rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, NULL);

  if (pcrow > 0) /* if 1 or more rows in the result set */
  {
    i = j = 0;

    printf("*****\n");
    printf("Orders for Customer: %ld\n", Ord[0].Cust_Num);
    printf("*****\n");

    while (i < pcrow)
    { printf("\nOrder #: %ld\n", Ord[i].Ord_Num);
      printf("      Product  Quantity      Price\n");
      printf("      -----\n");
      j = i;
      while (Ord[j].Ord_Num == Ord[i].Ord_Num)
      { printf("      %8ld %16.71f %12.21f\n",
                Ord[i].Prod_Num, Ord[i].Quant, Ord[i].Amount);
        i++;
      }
      printf("      %12.21f\n", Ord[j].Total);
    } /* end while */
  } /* end if */
}
while ( SQLMoreResults(hstmt) == SQL_SUCCESS);
/* ... */

```

Using large objects

The term large object and the generic acronym LOB see any type of large object.

There are three LOB data types:

- Binary large object (BLOB)
- Character large object (CLOB)
- Double-byte character large object (DBCLOB)

These LOB data types are represented symbolically as SQL_BLOB, SQL_CLOB, SQL_DBCLOB respectively. The LOB symbolic constants can be specified or returned on any of the DB2 ODBC functions that take in or return an SQL data type argument (for example, SQLBindParameter() and SQLDescribeCol()). See Table 4 on page 31 for a complete list of symbolic and default C symbolic names for SQL data types.

Although LOB values can be very large, they can be retrieved and manipulated in the application address space. However, the application might not need or want the entire LOB transferred from the database server into application memory. In many cases an application needs to select a LOB value and operate on pieces of it. Transfer of LOB data using the piece-wise sequential method provided by SQLGetData() and SQLPutData(), following the ODBC model, can be very expensive and time consuming. The application can reference an individual LOB value using a LOB locator.

Using LOB locators

A LOB locator is a mechanism that allows an application program to manipulate a large object value in an efficient, random access manner. The locator is a runtime concept: it is not a persistent type and is not stored in the database; it is a mechanism used to see a LOB value during a transaction and does not persist beyond the transaction in which it is created. Each LOB locator type has its own C data type (SQL_C_BLOB_LOCATOR, SQL_C_CLOB_LOCATOR, SQL_C_DBCLOB_LOCATOR) which enables transfer of LOB locator values to and from the database server.

A LOB locator is a simple token value that represents a single LOB value. A locator is not a reference to a column in a row, rather it references a large object value. An operation performed on a locator does not affect the original LOB value stored in the row. An application can retrieve a LOB locator into an application variable (using the SQLBindCol() or SQLGetData() function) and can then apply the following DB2 ODBC functions to the associated LOB value by using the locator:

- SQLGetLength() gets the length of a string a LOB locator represents.
- SQLGetPosition() gets the position of a search string within a source string where a LOB locator represents the source string. A LOB locator can also represent the search string.

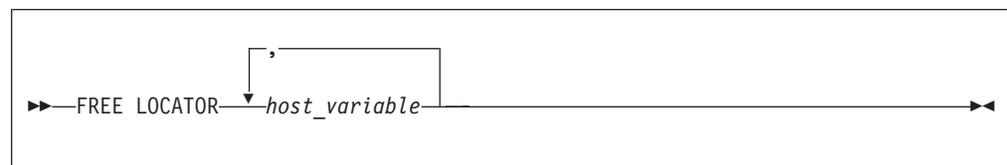
Locators are implicitly allocated by:

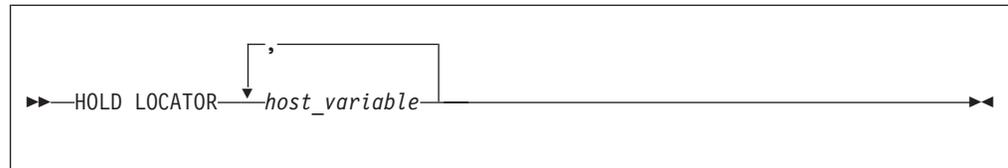
- Fetching a bound LOB column to the appropriate C locator type.
- Calling SQLGetSubString() and specifying that the substring be retrieved as a locator.
- Calling SQLGetData() on an unbound LOB column and specifying the appropriate C locator type. The C locator type must match the LOB column type or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

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

```
INSERT INTO TABLE4A
VALUES(1,CAST(? AS CLOB(2K)) CONCAT CAST(? AS CLOB(3K)))
```

The locator can be explicitly freed before the end of a transaction by executing the FREE LOCATOR statement or retained beyond a unit of work by executing the HOLD LOCATOR statement. The syntax is shown below.





Although this statement cannot be prepared dynamically, DB2 ODBC accepts it as a valid statement on `SQLPrepare()` and `SQLExecDirect()`. The application uses `SQLBindParameter()` with the SQL data type argument set to the appropriate SQL and C symbolic data types listed in Table 4 on page 31.

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.

LOB example

This example extracts the 'Interests' section from the Resume CLOB column of the `EMP_RESUME` table. Only the substring is transferred to the application.

```

/* ... */
SQLCHAR          stmt2[] =
                  "SELECT resume FROM emp_resume "
                  "WHERE empno = ? AND resume_format = 'ascii'";
/* ... */
/*****
** Get CLOB locator to selected Resume **
*****/
rc = SQLSetParam(hstmt, 1, SQL_C_CHAR, SQL_CHAR, 7,
                0, 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 #: %s\n %s\n", Empno.s, buffer);
/* ... */

```

Using distinct types

In addition to SQL data types (referred to as base SQL data types), you can define distinct types. A distinct type, or user-defined data type, shares its internal representation with a source type, but is considered to be a separate and incompatible type for most operations. Distinct types are created using the CREATE DISTINCT TYPE SQL statement. See *DB2 SQL Reference* for information about using this statement.

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

The following guidelines apply to distinct types:

- All SQL to C data type conversion rules that apply to the built-in type also apply to the distinct type.
- The distinct type has the same default C type as the built-in type.
- `SQLDescribeCol()` returns the built-in type information. The distinct type name can be obtained by calling `SQLColAttribute()` with the input descriptor type set to `SQL_COLUMN_DISTINCT_TYPE`.
- SQL predicates that involve parameter markers must be explicitly cast to the distinct type or base SQL data types. This is required since the application can only deal with the built-in types, so before any operation can be performed using the parameter, it must be cast from the C built-in type to the distinct type; otherwise an error occurs when the statement is prepared. For more information about casting distinct types, see “Casting distinct types” on page 455.

Distinct type example

This example shows the definition of distinct types and user-defined functions, and tables with distinct type columns. For an example that inserts rows into a table with distinct type columns, see “Array input example” on page 405.

```
/* ... */
/* 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 %ld Statements\n", num_stmts);

    /* Execute Direct statements */
    for (i = 0; i < num_stmts; i++) {
        rc = SQLExecDirect(hstmt, stmt[i], SQL_NTS);
    }
/* ... */

```

Using stored procedures

An application can be designed to run in two parts, one on the client and the other on the server. The stored procedure is a server application that runs at the database within the same transaction as the client application. Stored procedures can be written in either embedded SQL or using the DB2 ODBC functions (see "Writing a DB2 ODBC stored procedure" on page 419).

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. Any combination of embedded SQL and DB2 ODBC applications can be used. Figure 16 illustrates this concept.

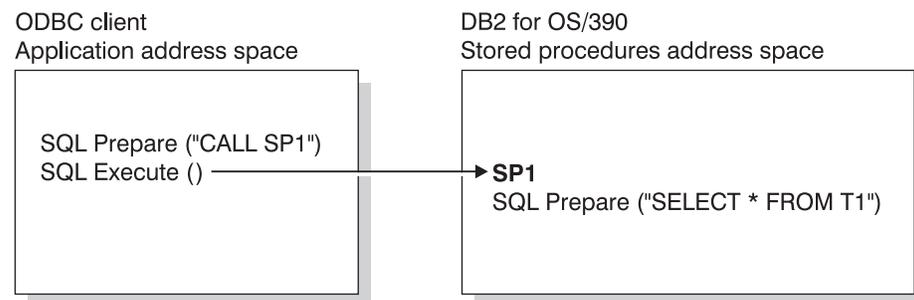


Figure 16. Running stored procedures

Advantages of using stored procedures

In general, stored procedures have the following advantages:

- Avoid network transfer of large amounts of data obtained as part of intermediate results in a long sequence of queries.
- Deployment of client database applications into client/server pieces.

In addition, stored procedures written in embedded static SQL have the following advantages:

- Performance - static SQL is prepared at precompile time and has no run time overhead of access plan (package) generation.
- Encapsulation (information hiding) - users do not need to know the details about 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 there is no need to grant explicit access to each database object. For example, you can grant a user run access for a stored procedure that selects data from tables for which the user does not have select privilege.

Catalog table for stored procedures

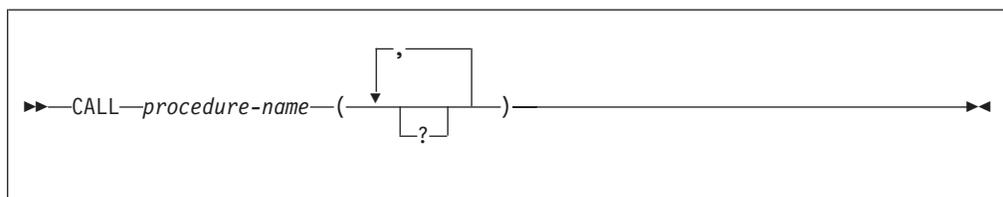
If the server is DB2 UDB Version 2 Release 1 or later, or DB2 for MVS/ESA Version 4 or later, an application can call `SQLProcedureColumns()` to determine the type of a parameter in a procedure call.

If the stored procedure resides on a DB2 for MVS/ESA Version 4 or a DB2 for OS/390 Version 5 server, the name of the stored procedure must be defined in the `SYSIBM.SYSPROCEDURES` catalog table. If the stored procedure resides on a DB2 for OS/390 Version 6 server or later, the name of the stored procedure must be defined in the `SYSIBM.SYSROUTINES` catalog table.

For more information, see “Catalog table for stored procedures” and “SQLProcedureColumns - Get input/output parameter information for a procedure” on page 313.

Calling stored procedures from a DB2 ODBC application

Stored procedures are invoked from a DB2 ODBC application by passing the following CALL statement syntax to `SQLExecDirect()` or to `SQLPrepare()` followed by `SQLExecute()`.



procedure-name

The name of the stored procedure to execute.

Although the CALL statement cannot be prepared dynamically, DB2 ODBC accepts the CALL statement as if it can be dynamically prepared. Stored procedures can also be called using the ODBC vendor escape sequence shown in “Stored procedure CALL syntax” on page 451.

For more information regarding the use of the CALL statement and stored procedures, see *DB2 SQL Reference* and *DB2 Application Programming and SQL Guide*.

If the server is DB2 UDB Version 2 Release 1 or later, or DB2 for MVS/ESA Version 4 or later, `SQLProcedures()` can be called to obtain a list of stored procedures available at the database.

The `?` in the CALL statement syntax diagram denotes parameter markers corresponding to the arguments for a stored procedure. All arguments must be passed using parameter markers; literals, the `NULL` keyword, and special registers are not allowed. However, literals can be used if a vendor escape clause for the CALL statement is used. See “Using vendor escape clauses” on page 448.

The parameter markers in the CALL statement are bound to application variables using `SQLBindParameter()`. Although stored procedure arguments can be used both for input and output, in order to avoid sending unnecessary data between the client and the server, the application should specify on `SQLBindParameter()` the parameter type of an input argument to be `SQL_PARAM_INPUT` and the parameter type of an output argument to be `SQL_PARAM_OUTPUT`. Those arguments that are both input and output have a parameter type of `SQL_PARAM_INPUT_OUTPUT`. Literals are considered type `SQL_PARAM_INPUT` only.

Writing a DB2 ODBC stored procedure

Although embedded SQL stored procedures provide the most advantages, application developers who have existing DB2 ODBC applications might wish to move components of the application to run on the server. In order to minimize the required changes to the code and logic of the application, these components can be implemented by writing stored procedures using DB2 ODBC.

Auto-commit must be off. This is achieved by using the `AUTOCOMMIT` keyword in the initialization file or by using the `SQLSetConnectAttr` API with the `SQL_AUTOCOMMIT` connect option `SQL_AUTOCOMMIT_OFF`.

`SQLConnect()` should be a null connect. Since all the internal information related to a DB2 ODBC connection is referenced by the connection handle, and since a stored procedure runs under the same connection and transaction as the client application, a stored procedure written using DB2 ODBC must make a null `SQLConnect()` call to associate a connection handle with the underlying connection of the client application. In a null `SQLConnect()` call, the `szDSN`, `szUID`, and `szAuthStr` argument pointers are all set to `NULL` and their respective length arguments all set to 0.

For stored procedures written in DB2 ODBC, the `COMMIT_ON_RETURN` option has no effect on DB2 ODBC rules; set it to 'N'. However, be aware that setting this option to 'N' overrides the manual-commit mode set in the client application.

The `MVSATTACHTYPE` keyword should be set to `RRSAF` if the stored procedure contains any LOB data types or distinct types in its parameter list. DB2 for OS/390 and z/OS requires that stored procedures containing any LOBs or distinct types must run in a WLM-established stored procedure address space. For further information on setting up the stored procedures environment, see Part 5 of *DB2 Application Programming and SQL Guide*.

For information about binding a stored procedure that runs under DB2 ODBC, see “Bind stored procedures” on page 45.

Returning result sets from stored procedures

DB2 ODBC provides the ability to retrieve one or more result sets from a stored procedure call, provided the stored procedure is coded such that one or more cursors, each associated with a query, is opened and left open when the stored procedure exits. If more than one cursor is left open, multiple result sets are returned.

Stored procedures written to return one or more result sets to a DB2 ODBC application should indicate the maximum number of result sets that can be returned in the `RESULT_SETS` column of the `SYSIBM.SYSPROCEDURES` table (DB2 for MVS/ESA Version 4 or DB2 for OS/390 Version 5) or the `RESULT_SETS` column of the `SYSIBM.SYSROUTINES` table (DB2 for OS/390 Version 6 or later). A zero in this column indicates that open cursors returned no result sets.

Programming stored procedures to return result sets

To return one or more result sets to a DB2 ODBC application the stored procedure must satisfy the following requirements:

- The stored procedure indicates that it wants a result set returned by declaring a cursor on the result set, opening a cursor on the result set (that is, executing the query), and leaving the cursor open when exiting the stored procedure.
- For every cursor that is left open, a result set is returned to the application.
- If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened in the stored procedure.
- In a stored procedure, DB2 ODBC uses a cursor declared `WITH RETURN`. If the cursor is closed before the stored procedure exit, it is a local cursor. If the cursor remains open upon stored procedure exit, it returns a query result set (also called a multiple result set) to the client application.
- To leave the cursor open to return result sets, the application must follow these guidelines:
 - Issue `SQLExecute()` or `SQLExecDirect()`.
 - Optionally, `SQLFetch()` rows.
 - Do not issue `SQLFreeHandle()` with *HandleType* set to `SQL_HANDLE_STMT` or `SQLCloseCursor()`.
 - Issue `SQLDisconnect()`, `SQLFreeHandle()` with *HandleType* set to `SQL_HANDLE_DBC`, and `SQLFreeHandle()` with *HandleType* set to `SQL_HANDLE_END` to terminate with the statement handle in a valid state.

By avoiding `SQLFreeHandle()` with *HandleType* set to `SQL_HANDLE_STMT`, the cursor remains open to return result sets. Appendix F, “Example code”, on page 507 provides an example; see case 2 of step 4.

- Only unread rows are passed back. For example, if the result set of a cursor has 500 rows, and 150 of those rows were already read by the stored procedure when it terminated, then rows 151 through 500 are returned to the stored procedure. This can be useful if the stored procedure wishes to filter out some initial rows and not return them to the application.

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. The following restrictions apply:

- Column names are not returned by either `SQLDescribeCol()` or `SQLColAttribute()` for static query statements. In this case, the ordinal position of the column is returned instead.
- All result sets are read-only.

- Schema functions (such as `SQLTables()`) cannot be used to return a result set. If schema functions are used within a stored procedure, all of the cursors for the associated statement handles must be closed before returning, otherwise extraneous result sets might be returned.
- When a query is prepared, result set column information is available before the execute. When a stored procedure is prepared, the result set column information is not available until the `CALL` statement is executed.

Programming DB2 ODBC client applications to receive result sets

DB2 ODBC applications can retrieve result sets after the execution of a stored procedure that leaves cursors open. The following guidelines explain the process and requirements.

- Before the stored procedure is called, ensure that there are no open cursors associated with the statement handle.
- Call the stored procedure.
- The execution of the stored procedure `CALL` statement effectively causes the cursors associated with the result sets to open.
- Examine any output parameters that are returned by the stored procedure. For example, the procedure might be designed so that there is an output parameter that indicates exactly how many result sets are generated.
- The DB2 ODBC application can then process a query as it normally does. If the application does not know the nature of the result set or the number of columns returned, it can call `SQLNumResultCols()`, `SQLDescribeCol()` or `SQLColAttribute()`. Next, the application can use any permitted combination of `SQLBindCol()`, `SQLFetch()`, and `SQLGetData()` to obtain the data in the result set.
- When `SQLFetch()` returns `SQL_NO_DATA_FOUND` or if the application is finished with the current result set, the application can call `SQLMoreResults()` to determine if there are more result sets to retrieve. Calling `SQLMoreResults()` closes the current cursor and advances processing to the next cursor that was left open by the stored procedure.
- If there is another result set, then `SQLMoreResults()` returns `SQL_SUCCESS`; otherwise, it returns an `SQL_NO_DATA_FOUND`.
- Result sets must be processed in serial fashion by the application, that is, one at a time in the order that they were opened in the stored procedure.

Stored procedure example with query result set

A detailed stored procedure example is provided in Appendix F, “Example code”, on page 507.

Writing multithreaded applications

This section explains DB2 ODBC’s support of multithreading and multiple contexts, and provides guidelines for programming techniques.

DB2 ODBC support of multiple LE threads

All DB2 ODBC applications have at least one LE thread created automatically in the application’s LE *enclave*. A multithreaded DB2 ODBC application creates additional LE threads using the *POSIX Pthread* function `pthread_create()`. These additional threads share the same reentrant copy of DB2 ODBC code within the LE enclave.

DB2 ODBC code is *reentrant* but uses shared storage areas that must be protected if multiple LE threads are running concurrently in the same enclave. The quality of

being reentrant and correctly handling shared storage areas is referred to as *threadsafe*. This quality is required by multithreaded applications.

DB2 ODBC supports concurrent execution of LE threads by making all of the DB2 ODBC function calls threadsafe. This threadsafe quality of function calls is only available if DB2 ODBC has access to OS/390 UNIX. DB2 ODBC uses the Pthread *mutex* functions of OS/390 UNIX to provide threadsafe function calls by serializing critical sections of DB2 ODBC code. See “Initialization keywords” on page 55 for a description of the THREADSAFE keyword.

Because OS/390 UNIX is present, threadsafe capability is available by default when executing a DB2 ODBC application in the following environments:

- The OS/390 UNIX shell
- TSO or batch for HFS-resident applications using the IBM supplied BPXBATCH program. (See *OS/390 UNIX System Services Command Reference* for more information about BPXBATCH).
- TSO or batch for applications that are not HFS-resident if the LE runtime option POSIX(ON) is specified when the application runs.

For example, to specify POSIX(ON) in TSO, you can invoke the DB2 ODBC application APP1 in the MVS dataset USER.RUNLIB.LOAD as follows:

```
CALL 'USER.RUNLIB.LOAD(APP1)' 'POSIX(ON)/'
```

Using batch JCL, you can invoke the same application:

```
//STEP1 EXEC PGM=APP1,PARM='POSIX(ON)/'  
//STEPLIB DD DSN=USER.RUNLIB.LOAD,DISP=SHR  
//          DD ...other libraries needed at runtime...
```

For more OS/390 UNIX information relating to DB2 ODBC, see “Special considerations for OS/390 UNIX” on page 50. Also, see *Language Environment for OS/390 & VM Programming Guide* for more information about running programs that use OS/390 UNIX.

Multithreaded applications allow threads to perform work in parallel on different connections as shown in Figure 17 on page 423.

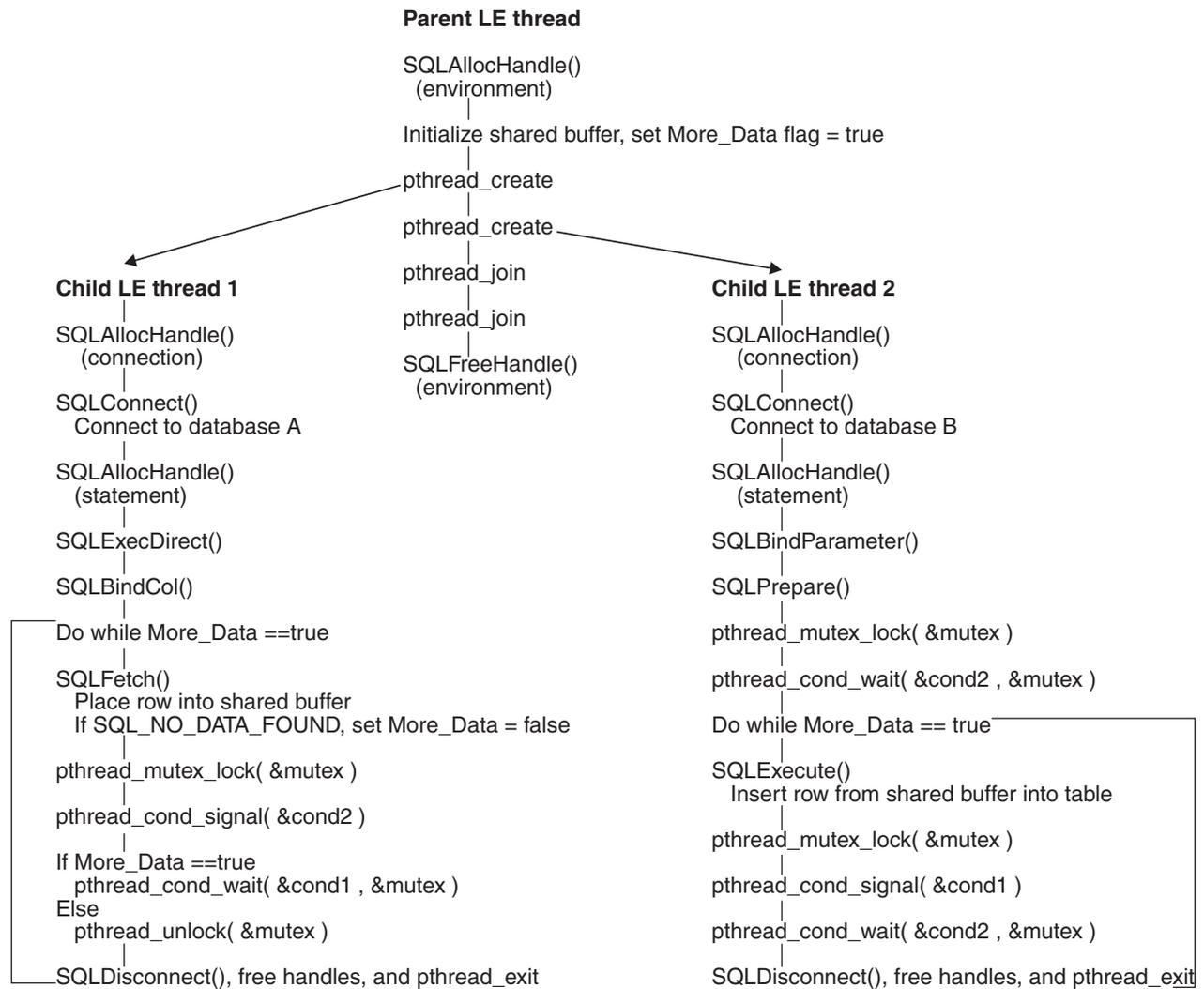


Figure 17. Multithreaded application

In Figure 17, an application implements a database-to-database copy as follows:

- A parent LE thread creates two child LE threads. The parent thread remains present for the duration of the activity of the child threads. DB2 ODBC requires that the thread which establishes the environment using `SQLAllocHandle()` must remain present for the duration of the application, so that DB2 language interface routines remain resident in the LE enclave.
- One child LE thread connects to database A and uses `SQLFetch()` calls to read data from one connection into a shared application buffer.
- Another child LE thread connects to database B and concurrently reads from the shared buffer, inserting the data into database B.
- Pthread functions are used to synchronize the use of the shared application buffer. See *OS/390 C/C++ Run-Time Library Reference* for a description of the Pthread functions.

When to use multiple LE threads

Detailed discussion of evaluating application requirements and making decisions about whether or not to use multithreading is beyond the scope of this book.

However, there are some general application types that are well-suited to multithreading. For example, applications that handle asynchronous work requests are candidates for multithreading.

An application that handles asynchronous work requests can take the form of a parent/child threading model in which the parent LE thread creates child LE threads to handle incoming work. The parent thread can then act as a dispatcher of these work requests as they arrive, directing them to child threads that are not currently busy servicing other work.

DB2 ODBC support of multiple contexts

The context consists of the application's logical connection to the data source and associated internal DB2 ODBC connection information that allows the application to direct its operations to a data source. In DB2 ODBC, the context is established when `SQLAllocHandle()` is issued with *HandleType* set to `SQL_HANDLE_DBC`.

The context is the DB2 ODBC equivalent of a DB2 thread. DB2 ODBC always creates a context when a successful `SQLAllocHandle()` (with *HandleType* set to `SQL_HANDLE_DBC`) is first issued by an application LE thread. If DB2 ODBC support for multiple contexts is not enabled, only the first `SQLAllocHandle()` for a LE thread establishes a context. With support for multiple contexts, DB2 ODBC establishes a separate context (and DB2 thread) each time the `SQLAllocHandle()` function is invoked.

If the initialization file specifies `MULTICONTEXT=0` (see "Initialization keywords" on page 55), there can only be one context for each LE thread that the application creates. This single context per thread can provide only the simulated support of the ODBC connection model, explained in "DB2 ODBC restrictions on the ODBC connection model" on page 17.

When the initialization file specifies `MULTICONTEXT=1`, a distinct context is established for each connection handle that is allocated when `SQLAllocHandle()` is issued. Using `MULTICONTEXT=1` requires:

- The RRSAF attachment facility, specified by `MVSATTACHTYPE=RRSAF` in the initialization file.
- OS/390 Unauthorized Context Services, available in OS/390 Version 2 Release 5 and higher releases.

Consistent with the ODBC connection model, the use of `MULTICONTEXT=1` implies `CONNECTTYPE=1`. The connections are independently handled by `SQLTransact` for both commit and rollback.

The creation of a context for each connection is consistent with, and provides full support for, the ODBC connection model.

The context is established with `SQLAllocHandle()` (with *HandleType* set to `SQL_HANDLE_DBC`) and deleted by `SQLFreeHandle()` (with *HandleType* set to `SQL_HANDLE_DBC`). All `SQLConnect()` and `SQLDisconnect()` operations that use the same connection handle belong to the same context. Although there can be only one active connection to a data source at any given time for the duration of the context, the target data source can be changed by `SQLDisconnect()` and `SQLConnect()`, subject to the rules of `CONNECTTYPE=1`.

With MULTICONTEXT=1 specified, DB2 ODBC automatically uses OS/390 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 are true:

- The DB2 ODBC application created a DB2 thread before invoking DB2 ODBC. This is always the case for a stored procedure using DB2 ODBC.
- The DB2 ODBC application created and switched to a private context before invoking DB2 ODBC. For example, an application that is explicitly using OS/390 Context Services and issues `ctxswch` to switch to a private context prior to invoking DB2 ODBC cannot take advantage of MULTICONTEXT=1.
- The DB2 ODBC application started a unit of recovery with any RRS resource manager before invoking DB2 ODBC.
- MVSATTACHTYPE=CAF is specified in the initialization file.
- The OS/390 operating system level does not support Unauthorized Context Services.

The application can use the `SQLGetInfo()` function with `infoType=SQL_MULTIPLE_ACTIVE_TXN` to determine if MULTICONTEXT=1 is active for the DB2 ODBC application. See “SQLGetInfo - Get general information” on page 234 for the description of `SQLGetInfo()`.

Table 169. Connection characteristics

Settings		Results		
MULTICONTEXT	CONNECTTYPE	Can LE thread have more than one ODBC connection with an outstanding unit of work?	Can LE thread commit/rollback ODBC connection independently?	Number of DB2 ODBC on behalf of application
0	2	Y	N	1 per LE thread
0	1	N	Y	1 per LE thread
1 ¹	1 or 2 ²	Y	Y	1 per ODBC connection handle

Note:

1. MULTICONTEXT=1 requires MVSATTACHTYPE=RRSAF and OS/390 Version 2 Release 5 or higher.
2. MULTICONTEXT=1 implies CONNECTTYPE=1 characteristics. With MULTICONTEXT=1 and CONNECTTYPE=2 specified in the initialization file, DB2 ODBC ignores CONNECTTYPE=2. With MULTICONTEXT=1 specified, any attempts to set CONNECTTYPE=2 using `SQLSetEnvAttr()`, `SQLSetConnectOptions()`, or `SQLDriverConnect()` are rejected with `SQLSTATE=01S02`.
 - All connections in a DB2 ODBC application have the same CONNECTTYPE and MULTICONTEXT characteristics. CONNECTTYPE is established at the first `SQLConnect()`. MULTICONTEXT is established at `SQLAllocHandle()` (with `HandleType` set to `SQL_HANDLE_ENV`).
 - For CONNECTTYPE=1 or MULTICONTEXT=1, the AUTOCOMMIT default is ON. For CONNECTTYPE=2 or MULTICONTEXT=0, the AUTOCOMMIT default is OFF.

Multiple contexts, one LE thread

When using the initialization file setting MULTICONTEXT=1, a DB2 ODBC application can create multiple independent connections for a LE thread. Figure 18 on page 426 is an example of a multicontext, one LE thread application.

```

/* 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 18. Example of independent connections on a single LE thread.

Multiple contexts, multiple LE threads

Using the initialization file setting MULTICONTEXT=1, combined with the default THREADSAFE=1, the application can create multiple independent connections under multiple LE threads. This capability can support complex DB2 ODBC server applications that handle multiple incoming work requests by using a fixed number of threads.

The multiple context, multiple LE thread capability requires some special considerations for the application using it. The Pthread functions should be used by the application for serialization of the use of connection handles and the associated statement handles. As an example of what can go wrong without proper serialization, see Figure 19 on page 427.

```

LE_Thread_1
.
.
.
rc = SQLExecDirect( hstmt1 , ... );
.
.
.
LE_Thread_2
.
.
.
SQLFreeHandle( hstmt1 , SQL_DROP);
.
.
.
rc = SQLExecDirect( hstmt1 , ... );
.
.
.
rc has the value SQL_INVALID_HANDLE because
LE_Thread_2 has freed statement handle hstmt1.

```

Figure 19. Example of improper serialization.

The suggested design is to map one LE thread per connection by establishing a pool of connections as shown in Figure 20.

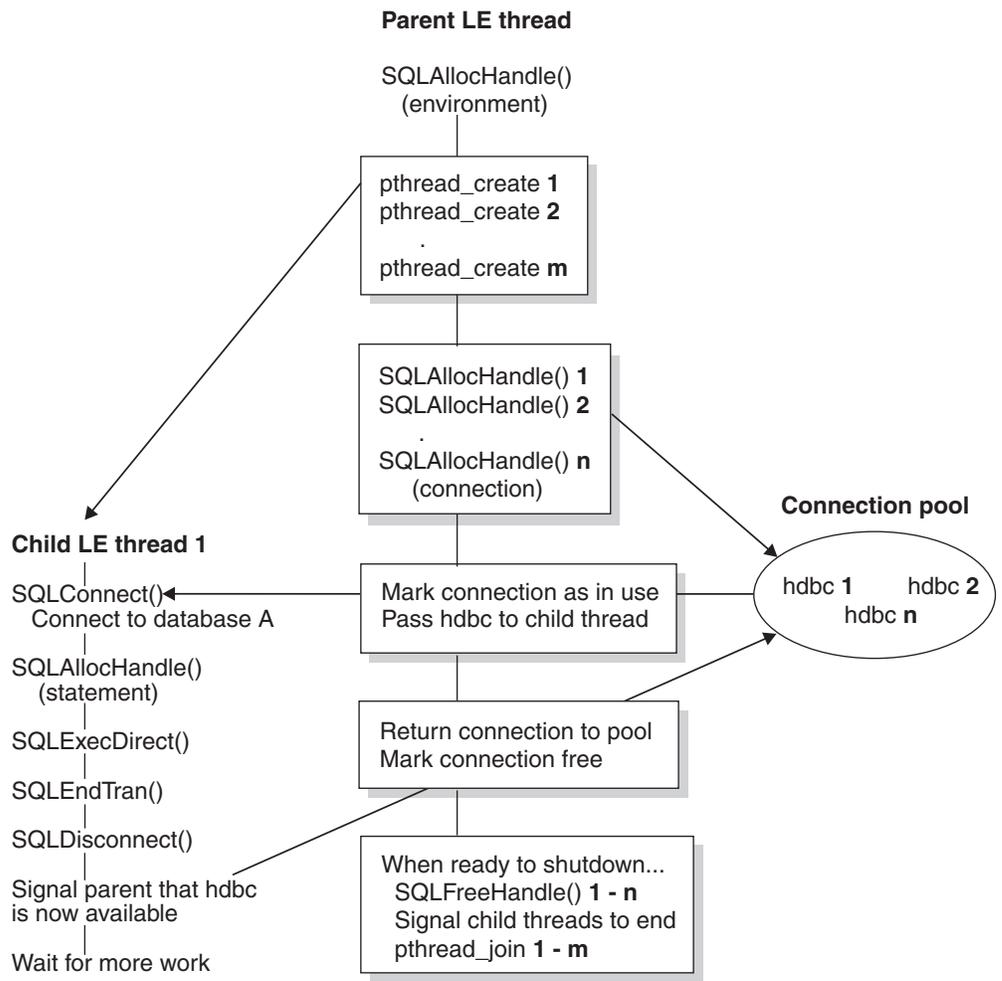


Figure 20. Model for multithreading with connection pooling (MULTICONTEXT=1)

In Figure 20, a pool of connections is established as follows:

- Designate a parent LE thread which allocates:

- m child LE threads
- n connection handles
- Each task that requires a connection is executed by one of the child LE threads, and is given one of the n connections by the parent LE thread. The parent thread remains active, acting as a dispatcher of tasks.
- The parent LE thread marks a connection as being in use until the child thread returns it to the connection pool.
- The parent LE thread frees the connections using `SQLFreeHandle()` when the parent thread is ending.

DB2 ODBC requires that the LE thread which establishes the environment using `SQLAllocHandle()` must remain present for the duration of the application, so that DB2 language interface routines will remain resident in the LE enclave.

This suggested design allows the parent LE thread to create more LE threads than connections if the threads are also used to perform non-SQL related tasks, or more connections than threads if the application should maintain a pool of active connections but limit the number of active tasks.

Connections can move from one application LE thread to another as the connections in the pool are assigned to child threads, returned to the pool, and assigned again.

The use of this design prevents two LE threads from trying to use the same connection (or an associated statement handle) at the same time. Although DB2 ODBC controls access to its internal resources, the application resources such as bound columns, parameter buffers, and files are not controlled by DB2 ODBC. If it is necessary for two threads to share an application resource, the application must implement some form of synchronization mechanism. For example, the database copy scenario in Figure 17 on page 423 uses Pthread functions to synchronize use of the shared buffer.

Application deadlocks

The possibility of timeouts, deadlocks, and general contention for database resources exists when multiple connections are used to access the same database resources concurrently.

An application that creates multiple connections by using multithreading or multiple context support can potentially create deadlocks with shared resources in the database.

A DB2 subsystem can detect deadlocks and rollback one or more connections to resolve them. An application can still deadlock if the following sequence occurs:

- Two LE threads connect to the same data source using two DB2 threads.
- One LE thread holds an internal application resource (such as a mutex) while its DB2 thread waits for access to a database resource.
- The other LE thread has a lock on a database resource while waiting for the internal application resource.

In this case the DB2 subsystem does not detect a deadlock since the application's internal resources cannot be monitored by a DB2 subsystem. However, the application is still subject to the DB2 subsystem detecting and handling any DB2 thread timeouts.

Using Unicode functions

This section describes the support for ODBC Unicode functions introduced in Version 6 of DB2 for OS/390 and z/OS.

Background

Unicode is a powerful encoding scheme that has the capacity to represent the characters of geographies and languages internationally.

The Unicode consortium published the Unicode standard. Extensive information about Unicode, the Unicode consortium, the Unicode standard, and standards conformance requirements is available at the following Web site: www.unicode.org.

The Microsoft ODBC 3.5 specification introduced ODBC Unicode functions that support the UCS-2 encoding form. UCS-2 is the Universal Character Set coded in 2 octets. Characters are represented in 16-bits for each character.

The ODBC Unicode functions are implemented as functions with a suffix of "W". With "suffix-W" functions, ODBC applications can:

- Pass Unicode UCS-2 SQL statements and data to a database server
- Retrieve data in Unicode UCS-2 from a database server

DB2 ODBC Unicode support

The DB2 for OS/390 and z/OS ODBC driver supports the UCS-2 encoding form with "suffix-W" APIs. The driver supports both suffix-W APIs and generic (non-suffixed) APIs that accept character string arguments or pointers to character string arguments. With this support, the driver can pass Unicode UCS-2 SQL statements and data streams to and from a DB2 for OS/390 and z/OS server (Version 6 and later). The driver performs the conversion of input and output character string arguments between Unicode UCS-2 and EBCDIC. The conversion is based on the EBCDIC CCSIDs that are set at DB2 installation time. For more information about CCSIDs set at installation time, see *DB2 Installation Guide*.

#

See "Setting up suffix-W API support" on page 46 for information about the setup required to use suffix-W API support.

Table 170 lists the suffix-W APIs that DB2 for OS/390 and z/OS supports and function syntax.

Table 170. Comparison of generic and suffix-W APIs

Generic APIs	Suffix-W APIs
SQLRETURN SQLColAttributes (SQLHSTMT hstmt, SQLUSMALLINT icol, SQLUSMALLINT fDescType, SQLPOINTER rgbDesc, SQLSMALLINT cbDescMax, SQLSMALLINT *pcbDesc, SQLINTEGER *pfDesc);	SQLRETURN SQLColAttributesW (SQLHSTMT hstmt, SQLUSMALLINT icol, SQLUSMALLINT fDescType, SQLPOINTER rgbDesc, SQLSMALLINT cbDescMax, SQLSMALLINT *pcbDesc, SQLINTEGER *pfDesc);

Table 170. Comparison of generic and suffix-W APIs (continued)

Generic APIs	Suffix-W APIs
SQLRETURN SQLColumns (SQLHSTMT hstmt SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLCHAR *szColumnsName, SQLSMALLINT cbColumnName);	SQLRETURN SQLColumnsW (SQLHSTMT hstmt SQLWCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLWCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLWCHAR *szTableName, SQLSMALLINT cbTableName, SQLWCHAR *szColumnsName, SQLSMALLINT cbColumnName);
SQLRETURN SQLColumnPrivileges (SQLHSTMT hstmt, SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLCHAR *szColumnsName, SQLSMALLINT cbColumnName);	SQLRETURN SQLColumnPrivilegesW (SQLHSTMT hstmt SQLWCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLWCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLWCHAR *szTableName, SQLSMALLINT cbTableName, SQLWCHAR *szColumnsName, SQLSMALLINT cbColumnName);
SQLRETURN SQLConnect (SQLHDBC hdbc, SQLCHAR *szDSN, SQLSMALLINT cbDSN, SQLCHAR *szUID, SQLSMALLINT cbUID, SQLCHAR *szAuthStr, SQLSMALLINT cbAuthStr);	SQLRETURN SQLConnectW (SQLHDBC hdbc, SQLWCHAR *szDSN, SQLSMALLINT cbDSN, SQLWCHAR *szUID, SQLSMALLINT cbUID, SQLWCHAR *szAuthStr, SQLSMALLINT cbAuthStr);
SQLRETURN SQLDataSources (SQLHENV henv, SQLSMALLINT fDirection, SQLCHAR *szDSN, SQLSMALLINT cbDSNMax, SQLSMALLINT *pcbDSN, SQLCHAR *szDescription, SQLSMALLINT cbDescriptionMax, SQLSMALLINT *pcbDescription);	SQLRETURN SQLDataSourcesW (SQLHENV henv, SQLSMALLINT fDirection, SQLWCHAR *szDSN, SQLSMALLINT cbDSNMax, SQLSMALLINT *pcbDSN, SQLWCHAR *szDescription, SQLSMALLINT cbDescriptionMax, SQLSMALLINT *pcbDescription);
SQLRETURN SQLDescribeCol (SQLHSTMT hstmt, SQLSMALLINT icol, SQLCHAR *szColName, SQLSMALLINT cbColNameMax, SQLSMALLINT *pcbColName, SQLSMALLINT *pfSqlType, SQLINTEGER *pcbColDef, SQLSMALLINT *pibScale, SQLSMALLINT *pfNullable);	SQLRETURN SQLDescribeColW (SQLHSTMT hstmt, SQLSMALLINT icol, SQLWCHAR *szColName, SQLSMALLINT cbColNameMax, SQLSMALLINT *pcbColName, SQLSMALLINT *pfSqlType, SQLINTEGER *pcbColDef, SQLSMALLINT *pibScale, SQLSMALLINT *pfNullable);
SQLRETURN SQLDriverConnect (SQLHDBC hdbc, SQLHWND hwnd, SQLCHAR *szConnStrIn, SQLSMALLINT cbConnStrIn, SQLCHAR *szConnStrOut, SQLSMALLINT cbConnStrOutMax, SQLSMALLINT pcbConnStrOut, SQLSMALLINT fDriverCompletion);	SQLRETURN SQLDriverConnectW (SQLHDBC hdbc, SQLHWND hwnd, SQLWCHAR *szConnStrIn, SQLSMALLINT cbConnStrIn, SQLWCHAR *szConnStrOut, SQLSMALLINT cbConnStrOutMax, SQLSMALLINT pcbConnStrOut, SQLSMALLINT fDriverCompletion);

Table 170. Comparison of generic and suffix-W APIs (continued)

Generic APIs	Suffix-W APIs
SQLRETURN SQLError (SQLHENV henv, SQLHDBC hdbc, SQLHSTMT hstmt, SQLCHAR *szSqlState, SQLINTEGER *pfNativeError, SQLCHAR *szErrorMsg, SQLSMALLINT cbErrorMsgMax, SQLSMALLINT *pcbErrorMsg);	SQLRETURN SQLErrorW (SQLHENV henv, SQLHDBC hdbc, SQLHSTMT hstmt, SQLWCHAR *szSqlState, SQLINTEGER *pfNativeError, SQLWCHAR *szErrorMsg, SQLSMALLINT cbErrorMsgMax, SQLSMALLINT *pcbErrorMsg);
SQLRETURN SQLExecDirect (SQLHSTMT hstmt, SQLCHAR *szSqlStr, SQLINTEGER cbSqlStr);	SQLRETURN SQLExecDirectW (SQLHSTMT hstmt, SQLWCHAR *szSqlStr, SQLINTEGER cbSqlStr);
SQLRETURN SQLForeignKeys (SQLHSTMT hstmt, SQLCHAR *szPkCatalogName, SQLSMALLINT :cbPkCatalogName, SQLCHAR *szPkSchemaName, SQLSMALLINT :cbPkSchemaName, SQLCHAR *szPkTableName, SQLSMALLINT :cbPkTableName, SQLCHAR *szFkCatalogName, SQLSMALLINT :cbFkCatalogName, SQLCHAR *szFkSchemaName, SQLSMALLINT :cbFkSchemaName, SQLCHAR *szFkTableName, SQLSMALLINT :cbFkTableName);	SQLRETURN SQLForeignKeysW (SQLHSTMT hstmt, SQLWCHAR *szPkCatalogName, SQLSMALLINT :cbPkCatalogName, SQLWCHAR *szPkSchemaName, SQLSMALLINT :cbPkSchemaName, SQLWCHAR *szPkTableName, SQLSMALLINT :cbPkTableName, SQLWCHAR *szFkCatalogName, SQLSMALLINT :cbFkCatalogName, SQLWCHAR *szFkSchemaName, SQLSMALLINT :cbFkSchemaName, SQLWCHAR *szFkTableName, SQLSMALLINT :cbFkTableName);
SQLRETURN SQLGetConnectOption (SQLHDBC hdbc, SQLUSMALLINT fOption, SQLINTEGER pvParam);	SQLRETURN SQLGetConnectOptionW (SQLHDBC hdbc, SQLUSMALLINT fOption, SQLINTEGER pvParam);
SQLRETURN SQLGetCursorName (SQLHSTMT hstmt, SQLCHAR *szCursor, SQLSMALLINT cbCursorMax, SQLSMALLINT *pcbCursor);	SQLRETURN SQLGetCursorNameW (SQLHSTMT hstmt, SQLWCHAR *szCursor, SQLSMALLINT cbCursorMax, SQLSMALLINT *pcbCursor);
SQLRETURN SQLGetInfo (SQLHDBC hdbc, SQLUSMALLINT fInfoType, SQLPOINTER rgbInfoValue, SQLSMALLINT cbInfoValueMax, SQLSMALLINT *pcbInfoValue);	SQLRETURN SQLGetInfoW (SQLHDBC hdbc, SQLUSMALLINT fInfoType, SQLPOINTER rgbInfoValue, SQLSMALLINT cbInfoValueMax, SQLSMALLINT *pcbInfoValue);
SQLRETURN SQLGetStmtOption (SQLHSTMT hstmt, SQLUSMALLINT fOption, SQLPOINTER pvParam);	SQLRETURN SQLGetStmtOptionW (SQLHSTMT hstmt, SQLUSMALLINT fOption, SQLPOINTER pvParam);
SQLRETURN SQLGetTypeInfo (SQLHSTMT hstmt, SQLSMALLINT fSqlType);	SQLRETURN SQLGetTypeInfoW (SQLHSTMT hstmt, SQLSMALLINT fSqlType);

Table 170. Comparison of generic and suffix-W APIs (continued)

Generic APIs	Suffix-W APIs
SQLRETURN SQLGetNativeSql (SQLHDBC hdbc, SQLCHAR *szSqlStrIn, SQLINTEGER cbSqlStrIn, SQLCHAR *szSqlStr, SQLINTEGER cbSqlStrMax, SQLINTEGER *pcbSqlStr);	SQLRETURN SQLGetNativeSqlW (SQLHDBC hdbc, SQLWCHAR *szSqlStrIn, SQLINTEGER cbSqlStrIn, SQLWCHAR *szSqlStr, SQLINTEGER cbSqlStrMax, SQLINTEGER *pcbSqlStr);
SQLRETURN SQLPrepare (SQLHSTMT hstmt, SQLCHAR *szSqlStr, SQLINTEGER cbSqlStr);	SQLRETURN SQLPrepareW (SQLHSTMT hstmt, SQLWCHAR *szSqlStr, SQLINTEGER cbSqlStr);
SQLRETURN SQLPrimaryKeys (SQLHSTMT hstmt, SQLCHAR *szCatalogName, SQLSMALLINT :cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT :cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT :cbTableName);	SQLRETURN SQLPrimaryKeysW (SQLHSTMT hstmt, SQLWCHAR *szCatalogName, SQLSMALLINT :cbCatalogName, SQLWCHAR *szSchemaName, SQLSMALLINT :cbSchemaName, SQLWCHAR *szTableName, SQLSMALLINT :cbTableName);
SQLRETURN SQLProcedureColumns (SQLHSTMT hstmt, SQLCHAR *szProcCatalog, SQLSMALLINT cbProcCatalog, SQLCHAR *szProcSchema, SQLSMALLINT cbProcSchema, SQLCHAR *szProcName, SQLSMALLINT cbProcName, SQLCHAR *szColumnName, SQLSMALLINT cbColumnName);	SQLRETURN SQLProcedureColumnsW (SQLHSTMT hstmt, SQLWCHAR *szProcCatalog, SQLSMALLINT cbProcCatalog, SQLWCHAR *szProcSchema, SQLSMALLINT cbProcSchema, SQLWCHAR *szProcName, SQLSMALLINT cbProcName, SQLWCHAR *szColumnName, SQLSMALLINT cbColumnName);
SQLRETURN SQLProcedures (SQLHSTMT hstmt, SQLCHAR *szProcCatalog, SQLSMALLINT cbProcCatalog, SQLCHAR *szProcSchema, SQLSMALLINT cbProcSchema, SQLCHAR *szProcName, SQLSMALLINT cbProcName);	SQLRETURN SQLProceduresW (SQLHSTMT hstmt, SQLWCHAR *szProcCatalog, SQLSMALLINT cbProcCatalog, SQLWCHAR *szProcSchema, SQLSMALLINT cbProcSchema, SQLWCHAR *szProcName, SQLSMALLINT cbProcName);
SQLRETURN SQLSetConnectOption (SQLHDBC hdbc, SQLSMALLINT fOption, SQLPOINTER pvParam);	SQLRETURN SQLSetConnectOptionW (SQLHDBC hdbc, SQLSMALLINT fOption, SQLPOINTER pvParam);
SQLRETURN SQLSetCursorName (SQLHSTMT hstmt, SQLCHAR *szCursor, SQLSMALLINT cbCursor);	SQLRETURN SQLSetCursorNameW (SQLHSTMT hstmt, SQLWCHAR *szCursor, SQLSMALLINT cbCursor);
SQLRETURN SQLSetStmtOption (SQLHSTMT hstmt, SQLSMALLINT fOption, SQLINTEGER pvParam);	SQLRETURN SQLSetStmtOptionW (SQLHSTMT hstmt, SQLSMALLINT fOption, SQLINTEGER pvParam);

Table 170. Comparison of generic and suffix-W APIs (continued)

Generic APIs	Suffix-W APIs
SQLRETURN SQLSpecialColumns (SQLHSTMT hstmt SQLUSMALLINT fColType, SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLUSMALLINT fScope, SQLUSMALLINT fNulltable);	SQLRETURN SQLSpecialColumnsW (SQLHSTMT hstmt SQLUSMALLINT fColType, SQLWCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLWCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLWCHAR *szTableName, SQLSMALLINT cbTableName, SQLUSMALLINT fScope, SQLUSMALLINT fNulltable);
SQLRETURN SQLStatistics (SQLHSTMT hstmt SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLUSMALLINT fUnique, SQLUSMALLINT fAccuracy);	SQLRETURN SQLStatisticsW (SQLHSTMT hstmt SQLWCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLWCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLWCHAR *szTableName, SQLSMALLINT cbTableName, SQLUSMALLINT fUnique, SQLUSMALLINT fAccuracy);
SQLRETURN SQLTablePrivileges (SQLHSTMT hstmt SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName);	SQLRETURN SQLTablePrivilegesW (SQLHSTMT hstmt SQLWCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLWCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLWCHAR *szTableName, SQLSMALLINT cbTableName);
SQLRETURN SQLTables (SQLHSTMT hstmt SQLCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLCHAR *szTableName, SQLSMALLINT cbTableName, SQLCHAR *szTableType, SQLSMALLINT cbTableType);	SQLRETURN SQLTablesW (SQLHSTMT hstmt SQLWCHAR *szCatalogName, SQLSMALLINT cbCatalogName, SQLWCHAR *szSchemaName, SQLSMALLINT cbSchemaName, SQLWCHAR *szTableName, SQLSMALLINT cbTableName, SQLWCHAR *szTableType, SQLSMALLINT cbTableType);

Applications that make calls to suffix-W APIs must be running on OS/390 Release 8 or later.

Application programming guidelines

This section describes programming considerations for DB2 ODBC applications that use suffix-W functions and provides a sample program.

ODBC C data types

ODBC applications that bind Unicode UCS-2 application buffers must use the ODBC C data type, `SQL_C_WCHAR`. You can specify `SQL_C_WCHAR` in the *fCType* argument of the following APIs:

- `SQLBindCol`
- `SQLBindParameter`
- `SQLGetData`
- `SQLSetParam`

For example, you can use `SQLBindCol` to bind the first column of a result set to a Unicode UCS-2 application buffer:

```
SQLBindCol( (SQLHSTMT) hstmt,  
            (SQLUSMALLINT) 1,  
            (SQLSMALLINT) SQL_C_WCHAR,  
            (SQLPOINTER) UCSWSTR,  
            (SQLINTEGER) sizeof(UCSWSTR),  
            (SQLINTEGER *) &LEN_UCSWSTR );
```

ODBC C type definition

Applications must use the ODBC C type definition, `SQLWCHAR`, when declaring variables or arguments used with suffix-W functions.

For example, you can specify a Unicode UCS-2 argument on a suffix-W API call as follows:

```
SQLExecDirectW( (SQLHSTMT) hstmt,  
                (SQLWCHAR *) UCSWSTR,  
                (SQLINTEGER) SQL_NTS );
```

Binding application variables to parameter markers

For UCS-2 application variables, applications must bind parameter markers to the `SQL_C_WCHAR` data type.

Retrieving result set data into application variables

An ODBC application variable can be associated with a column of a DB2 result set using the `SQLBindCol` or `SQLGetData` APIs. Applications that expect to retrieve any character or graphic data in UCS-2 encoding, must use the `SQL_C_WCHAR` symbolic C data type when binding data buffers.

Length of string arguments

For UCS-2 character strings:

- The length argument is always passed as count-of-characters
- Specifying `SQL_NTS` indicates that the UCS-2 string is null-terminated and includes a two-byte null terminator
- Specify the value `SQL_NULL_DATA` to pass a null value.

For more information about working with string arguments in DB2 ODBC functions, see “Working with string arguments” on page 35.

Example: ODBC application using suffix-W APIs

The following example shows an ODBC application that uses three suffix-W APIs.

```

/*****
/* Main program
/* - CREATE TABLE 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 <sqlca.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  pcpwr=0;
SQLSMALLINT  pccol=0;

SQLCHAR      sqlstmt[200];
SQLINTEGER   sqlstmtlen;

SQLWCHAR     H1INT4 [50 ];
SQLWCHAR     H1SMINT [50 ];
SQLWCHAR     H1CHR10 [50 ];
SQLWCHAR     H1CHR10MIX [50 ];
SQLWCHAR     H1VCHR20 [50 ];
SQLWCHAR     H1VCHR20MIX [50 ];
SQLWCHAR     H1GRA10 [50 ];
SQLWCHAR     H1VGRA20 [50 ];
SQLWCHAR     H1TTIME [50 ];
SQLWCHAR     H1DDATE [50 ];
SQLWCHAR     H1TSTMP [50 ];
SQLWCHAR     H2INT4 [50 ];

```

Figure 21. ODBC application using suffix-W APIs (Part 1 of 12)

```

SQLWCHAR      H2SMINT      [50 ] ;
SQLWCHAR      H2CHR10     [50 ] ;
SQLWCHAR      H2CHR10MIX [50 ] ;
SQLWCHAR      H2VCHR20    [50 ] ;
SQLWCHAR      H2VCHR20MIX [50 ] ;
SQLWCHAR      H2GRA10     [50 ] ;
SQLWCHAR      H2VGRA20    [50 ] ;
SQLWCHAR      H2TTIME     [50 ] ;
SQLWCHAR      H2DDATE     [50 ] ;
SQLWCHAR      H2TSTMP     [50 ] ;

SQLINTEGER    LEN_H1INT4 ;
SQLINTEGER    LEN_H1SMINT ;
SQLINTEGER    LEN_H1CHR10 ;
SQLINTEGER    LEN_H1CHR10MIX ;
SQLINTEGER    LEN_H1VCHR20 ;
SQLINTEGER    LEN_H1VCHR20MIX ;
SQLINTEGER    LEN_H1GRA10 ;
SQLINTEGER    LEN_H1VGRA20 ;
SQLINTEGER    LEN_H1TTIME ;
SQLINTEGER    LEN_H1DDATE ;
SQLINTEGER    LEN_H1TSTMP ;

SQLINTEGER    LEN_H2INT4 ;
SQLINTEGER    LEN_H2SMINT ;
SQLINTEGER    LEN_H2CHR10 ;
SQLINTEGER    LEN_H2CHR10MIX ;
SQLINTEGER    LEN_H2VCHR20 ;
SQLINTEGER    LEN_H2VCHR20MIX ;
SQLINTEGER    LEN_H2GRA10 ;
SQLINTEGER    LEN_H2VGRA20 ;
SQLINTEGER    LEN_H2TTIME ;
SQLINTEGER    LEN_H2DDATE ;
SQLINTEGER    LEN_H2TSTMP ;

SQLWCHAR      DROPW1      [100] ;
SQLWCHAR      DELETEW1    [100] ;
SQLWCHAR      SELECTW1    [100] ;
SQLWCHAR      CREATEW1    [500] ;
SQLWCHAR      INSERTW1    [500] ;

SQLWCHAR      DROPW2      [100] ;
SQLWCHAR      DELETEW2    [100] ;
SQLWCHAR      SELECTW2    [100] ;
SQLWCHAR      CREATEW2    [500] ;
SQLWCHAR      INSERTW2    [500] ;

SQLINTEGER    LEN_H1INT4 ;
SQLINTEGER    LEN_DROPW1 ;
SQLINTEGER    LEN_DELETEW1 ;
SQLINTEGER    LEN_INSERTW1 ;
SQLINTEGER    LEN_CREATEW1 ;
SQLINTEGER    LEN_SELECTW1 ;

```

Figure 21. ODBC application using suffix-W APIs (Part 2 of 12)

```

SQLINTEGER      LEN_DROPW2;
SQLINTEGER      LEN_DELETEW2;
SQLINTEGER      LEN_INSERTW2;
SQLINTEGER      LEN_CREATEW2;
SQLINTEGER      LEN_SELECTW2;

struct {
    short LEN;
    char DATA[200]; } STMTSQL;

long            SPCODE;
int             result;
int             ix, locix;

/*****

int main()
{

    henv=0;
    rc = SQLA1locEnv(&henv);
    if( rc != SQL_SUCCESS ) goto dberror;

    hdbc=0;
    rc=SQLA1locConnect(henv, &hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;

    /*****
    /* Setup application host variables (UCS-2 character strings) */
    /*****

#pragma convlit(resume)

    wcscpy(uid, (wchar_t *)"jgold");
    wcscpy(pwd, (wchar_t *)"general");
    wcscpy(server, (wchar_t *)"STLEC1");
    wcscpy(DROPW1, (wchar_t *)
        "DROP TABLE MYTABLE");
    LEN_DROPW1=wcslen((wchar_t *)DROPW1);

    wcscpy(SELECTW1, (wchar_t *)
        "SELECT * FROM MYTABLE WHERE INT4=200");
    LEN_SELECTW1=wcslen((wchar_t *)SELECTW1);

    wcscpy(CREATEW1, (wchar_t *)
        "CREATE TABLE MYTABLE ( ");
    wcscat(CREATEW1, (wchar_t *)
        "INT4 INTEGER, SMINT SMALLINT, ");
    wcscat(CREATEW1, (wchar_t *)
        "CHR10 CHAR(10), CHR10MIX CHAR(10) FOR MIXED DATA, ");

```

Figure 21. ODBC application using suffix-W APIs (Part 3 of 12)

```

wscat(CREATEW1, (wchar_t *)
    "VCHR20 VARCHAR(20), VCHR20MIX VARCHAR(20) FOR MIXED DATA, ");
wscat(CREATEW1, (wchar_t *)
    "GRA10 GRAPHIC(10), VGRA20 VARGRAPHIC(20), ");
wscat(CREATEW1, (wchar_t *)
    "TTIME TIME, DDATE DATE, TSTMP TIMESTAMP )" );
LEN_CREATEW1=wcslen((wchar_t *)CREATEW1);

wscpy(DELETEW1, (wchar_t *)
    "DELETE FROM MYTABLE WHERE INT4 IS NULL OR INT4 IS NOT NULL");
LEN_DELETEW1=wcslen((wchar_t *)DELETEW1);
wscpy(INSERTW1, (wchar_t *)
    "INSERT INTO MYTABLE VALUES ( ");
wscat(INSERTW1, (wchar_t *)
    "( 100,1,'CHAR10','CHAR10MIX','VARCHAR20','VARCHAR20MIX', ");
wscat(INSERTW1, (wchar_t *)
    "G' A B C', VARGRAPHIC('ABC'), ");
wscat(INSERTW1, (wchar_t *)
    "'3:45 PM', '06/12/1999', ");
wscat(INSERTW1, (wchar_t *)
    "'1999-09-09-09.09.09.090909' )" );
LEN_INSERTW1=wcslen((wchar_t *)INSERTW1);

wscpy(INSERTW2, (wchar_t *)
    "INSERT INTO MYTABLE VALUES (?,?,?,?,?,?,?,?,?,?)");
LEN_INSERTW2=wcslen((wchar_t *)INSERTW2);

wscpy(H1INT4      , (wchar_t *)"200");
wscpy(H1SMINT    , (wchar_t *)"5");
wscpy(H1CHR10    , (wchar_t *)"CHAR10");
wscpy(H1CHR10MIX , (wchar_t *)"CHAR10MIX");
wscpy(H1VCHR20   , (wchar_t *)"VARCHAR20");
wscpy(H1VCHR20MIX, (wchar_t *)"VARCHAR20MIX");
wscpy(H1TTIME    , (wchar_t *)"3:45 PM");
wscpy(H1DDATE    , (wchar_t *)"06/12/1999");
wscpy(H1TSTMP   , (wchar_t *)"1999-09-09-09.09.09.090909");

#pragma convlit(suspend)
/* 0xFF21,0xFF22,0xFF23,0x0000 */
wscpy(H1GRA10    , (wchar_t *)"    ");
/* 0x0041,0xFF21,0x0000 */
wscpy(H1VGRA20   , (wchar_t *)"    ");

LEN_H1INT4      = SQL_NTS;
LEN_H1SMINT     = SQL_NTS;
LEN_H1CHR10     = SQL_NTS;
LEN_H1CHR10MIX = SQL_NTS;
LEN_H1VCHR20    = SQL_NTS;
LEN_H1VCHR20MIX = SQL_NTS;
LEN_H1GRA10     = SQL_NTS;
LEN_H1VGRA20    = SQL_NTS;
LEN_H1TTIME     = SQL_NTS;
LEN_H1DDATE     = SQL_NTS;
LEN_H1TSTMP    = SQL_NTS;

```

Figure 21. ODBC application using suffix-W APIs (Part 4 of 12)

```

/*****
/* SQLConnectW
/*****
rc=SQLConnectW(hdbc, NULL, 0, NULL, 0, NULL, 0);
if( rc != SQL_SUCCESS ) goto dberror;

/*****
/* DROP TABLE - SQLExecuteDirectW
/*****
hstmt=0;
rc=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;

rc=SQLExecDirectW(hstmt,DROPW1,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;

rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

rc=SQLFreeStmt(hstmt, SQL_DROP);
if( rc != SQL_SUCCESS ) goto dberror;
/*****
/* CREATE TABLE MYTABLE - SQLPrepareW
/*****
hstmt=0;
rc=SQLAllocStmt(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=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

rc=SQLFreeStmt(hstmt, SQL_DROP);
if( rc != SQL_SUCCESS) goto dberror;
/*****
/* INSERT INTO MYTABLE with literals - SQLExecDirectW
/*****
hstmt=0;
rc=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
rc=SQLExecDirectW(hstmt,DROPW1,SQL_NTS);
if( rc != SQL_SUCCESS) goto dberror;
rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS) goto dberror;
rc=SQLFreeStmt(hstmt, SQL_DROP);
if( rc != SQL_SUCCESS) goto dberror;

```

Figure 21. ODBC application using suffix-W APIs (Part 5 of 12)

```

/*****
/* INSERT INTO MYTABLE with parameter markers
/* - SQLPrepareW
/* - SQLBindParameter with SQL_C_WCHAR symbolic C data type
*****/
hstmt=0;
rc=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;

/* INSERT INTO MYTABLE VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?) */

rc=SQLPrepareW(hstmt, INSERTW2, SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;

rc=SQLNumParams(hstmt, &pcpar);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPDV1                                number= 19");
if( pcpar != 11 ) goto dberror;

/* Bind INTEGER parameter */
rc= SQLBindParameter(hstmt,
                    1,
                    SQL_PARAM_INPUT,
                    SQL_C_WCHAR,
                    SQL_INTEGER,
                    10,
                    0,
                    (SQLPOINTER)H1INT4,
                    sizeof(H1INT4),
                    (SQLINTEGER *)&LEN_H1INT4 );
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind SMALLINT parameter */
rc = SQLBindParameter(hstmt,
                    2,
                    SQL_PARAM_INPUT,
                    SQL_C_WCHAR,
                    SQL_SMALLINT,
                    5,
                    0,
                    (SQLPOINTER)H1SMINT,
                    sizeof(H1SMINT),
                    (SQLINTEGER *)&LEN_H1SMINT);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind CHAR(10) parameter */
rc = SQLBindParameter(hstmt,
                    3,
                    SQL_PARAM_INPUT,
                    SQL_C_WCHAR,
                    SQL_CHAR,
                    10,

```

Figure 21. ODBC application using suffix-W APIs (Part 6 of 12)

```

        0,
        (SQLPOINTER)H1CHR10,
        sizeof(H1CHR10),
        (SQLINTEGER *)&LEN_H1CHR10);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind CHAR(10) parameter */

rc = SQLBindParameter(hstmt,
    3,
    SQL_PARAM_INPUT,
    SQL_C_WCHAR,
    SQL_CHAR,
    10,
    0,
    (SQLPOINTER)H1CHR10,
    sizeof(H1CHR10),
    (SQLINTEGER *)&LEN_H1CHR10);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind CHAR(10) FOR MIXED parameter */

rc = SQLBindParameter(hstmt,
    4,
    SQL_PARAM_INPUT,
    SQL_C_WCHAR,
    SQL_CHAR,
    10,
    0,
    (SQLPOINTER)H1CHR10MIX,
    sizeof(H1CHR10MIX),
    (SQLINTEGER *)&LEN_H1CHR10MIX);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind VARCHAR(20) parameter */

rc = SQLBindParameter(hstmt,
    5,
    SQL_PARAM_INPUT,
    SQL_C_WCHAR,
    SQL_VARCHAR,
    20,
    0,
    (SQLPOINTER)H1VCHR20,
    sizeof(H1VCHR20),
    (SQLINTEGER *)&LEN_H1VCHR20);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind VARCHAR(20) FOR MIXED parameter */

rc = SQLBindParameter(hstmt,
    6,
    SQL_PARAM_INPUT,
    SQL_C_WCHAR,
    SQL_VARCHAR,

```

Figure 21. ODBC application using suffix-W APIs (Part 7 of 12)

```

                20,
                0,
                (SQLPOINTER)H1VCHR20MIX,
                sizeof(H1VCHR20MIX),
                (SQLINTEGER *)&LEN_H1VCHR20MIX);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind GRAPHIC(10) parameter */

rc = SQLBindParameter(hstmt,
                    7,
                    SQL_PARAM_INPUT,
                    SQL_C_WCHAR,
                    SQL_GRAPHIC,
                    10,
                    0,
                    (SQLPOINTER)H1GRA10,
                    sizeof(H1GRA10),
                    (SQLINTEGER *)&LEN_H1GRA10);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind VARGRAPHIC(20) parameter*/

rc = SQLBindParameter(hstmt,
                    8,
                    SQL_PARAM_INPUT,
                    SQL_C_WCHAR,
                    SQL_VARGRAPHIC,
                    20,
                    0,
                    (SQLPOINTER)H1VGRA20,
                    sizeof(H1VGRA20),
                    (SQLINTEGER *)&LEN_H1VGRA20);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind TIME parameter */

rc= SQLBindParameter(hstmt,
                    9,
                    SQL_PARAM_INPUT,
                    SQL_C_WCHAR,
                    SQL_TIME,
                    8,
                    0,
                    (SQLPOINTER)H1TTIME,
                    sizeof(H1TTIME),
                    (SQLINTEGER *)&LEN_H1TTIME);
if( rc != SQL_SUCCESS) goto dberror;

/* Bind DATE parameter */

rc = SQLBindParameter(hstmt,
                    10,
                    SQL_PARAM_INPUT,

```

Figure 21. ODBC application using suffix-W APIs (Part 8 of 12)

```

                SQL_C_WCHAR,
                SQL_DATE,
                10,
                0,
                (SQLPOINTER)H1DDATE,
                sizeof(H1DDATE),
                (SQLINTEGER *)&LEN_H1DDATE);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind TIMESTAMP parameter */
rc = SQLBindParameter(hstmt,
                    11
                    SQL_PARAM_INPUT,
                    SQL_C_WCHAR,
                    SQL_DATE,
                    26,
                    0,
                    (SQLPOINTER)H1TSTMP,
                    sizeof(H1TSTMP),
                    (SQLINTEGER *)&LEN_H1TSTMP);

if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPDV1 SQLExecute                number= 25");
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS) goto dberror;

printf("\nAPDV1 SQLTransact                number=26");
rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPDV1 SQLFreeStmt                number= 27");
rc=SQLFreeStmt(hstmt, SQL_DROP);
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=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind INTEGER column */
rc = SQLBindCol(hstmt,
                1,
                SQL_C_WCHAR,
                (SQLPOINTER)H2INT4,
                sizeof(H2INT4 ),
                (SQLINTEGER *)&LEN_H2INT4 );
if( rc != SQL_SUCCESS ) goto dberror;

```

Figure 21. ODBC application using suffix-W APIs (Part 9 of 12)

```

/* Bind SMALLINT column */
rc = SQLBindCol(hstmt,
                2,
                SQL_C_WCHAR,
                (SQLPOINTER)H2SMINT,
                sizeof(H2SMINT),
                (SQLINTEGER *)&LEN_H2SMINT);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind CHAR(10) column */
rc = SQLBindCol(hstmt,
                3,
                SQL_C_WCHAR,
                (SQLPOINTER)H2CHR10,
                sizeof(H2CHR10),
                (SQLINTEGER *)&LEN_H2CHR10);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind CHAR(10) FOR MIXED column */
rc = SQLBindCol(hstmt,
                4,
                SQL_C_WCHAR,
                (SQLPOINTER)H2CHR10MIX,
                sizeof(H2CHR10MIX),
                (SQLINTEGER *)&LEN_H2CHR10MIX);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind VARCHAR(20) column */
rc = SQLBindCol(hstmt,
                5,
                SQL_C_WCHAR,
                (SQLPOINTER)H2VCHR20,
                sizeof(H2VCHR20),
                (SQLINTEGER *)&LEN_H2VCHR20);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind VARCHAR(20) FOR MIXED column */
rc = SQLBindCol(hstmt,
                6,
                SQL_C_WCHAR,
                (SQLPOINTER)H2VCHR20MIX,
                sizeof(H2VCHR20MIX),
                (SQLINTEGER *)&LEN_H2VCHR20MIX);
if( rc != SQL_SUCCESS ) goto dberror;

```

Figure 21. ODBC application using suffix-W APIs (Part 10 of 12)

```

/* Bind GRAPHIC(10) column */
rc = SQLBindCol(hstmt,
                7,
                SQL_C_WCHAR,
                (SQLPOINTER)H2GRA10,
                sizeof(H2GRA10),
                (SQLINTEGER *)&LEN_H2GRA10);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind VARGRAPHIC(20) column */
rc = SQLBindCol(hstmt,
                8,
                SQL_C_WCHAR,
                (SQLPOINTER)H2VGRA20,
                sizeof(H2VGRA20),
                (SQLINTEGER *)&LEN_H2VGRA20);
if( rc != SQL_SUCCESS ) goto dberror;
/* Bind TIME column */

rc = SQLBindCol(hstmt,
                9,
                SQL_C_WCHAR,
                (SQLPOINTER)H2TTIME,
                sizeof(H2TTIME),
                (SQLINTEGER *)&LEN_H2TTIME);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind DATE column */

rc = SQLBindCol(hstmt,
                10,
                SQL_C_WCHAR,
                (SQLPOINTER)H2DDATE,
                sizeof(H2DDATE),
                (SQLINTEGER *)&LEN_H2DDATE);
if( rc != SQL_SUCCESS ) goto dberror;

/* Bind TIMESTAMP column */

rc = SQLBindCol(hstmt,
                11,
                SQL_C_WCHAR,
                (SQLPOINTER)H2TSTMP,
                sizeof(H2TSTMP),
                (SQLINTEGER *)&LEN_H2TSTMP);
if( rc != SQL_SUCCESS ) goto dberror;
/*
 * SELECT * FROM MYTABLE WHERE INT4=200
 */

```

Figure 21. ODBC application using suffix-W APIs (Part 11 of 12)

```

rc=SQLExecDirectW(hstmt,SELECTW1,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;

rc=SQLFetch(hstmt);
if( rc != SQL_SUCCESS ) goto dberror;

rc=SQLFreeStmt(hstmt, SQL_DROP);
if( rc != SQL_SUCCESS ) goto dberror;

/*****/

rc=SQLDisconnect(hdbc);
if( rc != SQL_SUCCESS ) goto dberror;

rc=SQLFreeConnect(hdbc);
if( rc != SQL_SUCCESS ) goto dberror;

rc=SQLFreeEnv(henv);
if( rc != SQL_SUCCESS ) goto dberror;

dberror:
rc = SQL_ERROR;

return(rc);
} /*END MAIN*/

```

Figure 21. ODBC application using suffix-W APIs (Part 12 of 12)

Mixing embedded SQL and DB2 ODBC

It is possible, and sometimes desirable for an application to use DB2 ODBC in conjunction with embedded static SQL. Consider the scenario where the application developer wishes to take advantage of the ease of use provided by the DB2 ODBC catalog functions and maximize the portion of the application's processing where performance is critical. In order to mix the use of DB2 ODBC and embedded SQL, the application must comply to the following rules:

- All connection management and transaction management must be performed completely using either DB2 ODBC or embedded SQL. Either the DB2 ODBC application performs all the connects and commits/rollback and calls functions written using embedded SQL; or an embedded SQL application performs all the connects and commits/rollback and calls functions written in DB2 ODBC which use a null connection (see "Writing a DB2 ODBC stored procedure" on page 419 for details on null connections).
- Query statement processing must not and cannot straddle across DB2 ODBC and embedded SQL interfaces for the same statement; for example, the application cannot open a cursor in an embedded SQL routine, and then call the DB2 ODBC `SQLFetch()` function to retrieve row data.

Since DB2 ODBC permits multiple connections, the `SQLSetConnection()` function must be called prior to making a function call to a routine written in embedded SQL. This allows the application to explicitly specify the connection under which the embedded SQL routine should perform its processing. If the application only ever sets up one connection, or if the application is written entirely in DB2 ODBC, then calls to `SQLSetConnection()` are not needed.

Mixed embedded SQL and DB2 ODBC example

The following example demonstrates 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 there are two active connections */

/* set current connection to the first database */
if ( (rc = SQLSetConnection(hdbc[0])) != SQL_SUCCESS )
    printf("Error setting connection 1\n");

/* call function that contains embedded SQL */
if ((rc = Create_Tab() ) != 0)
    printf("Error Creating Table on 1st connection, RC=%ld\n", rc);

/* Commit transaction on connection 1 */
SQLEndTran(SQL_HANDLE_DBC, hdbc[0], SQL_COMMIT);

/* set current connection to the second database */
if ( (rc = SQLSetConnection(hdbc[1])) != SQL_SUCCESS )
    printf("Error setting connection 2\n");

/* call function that contains embedded SQL */
if ((rc = Create_Tab() ) != 0)
    printf("Error Creating Table on 2nd connection, RC=%ld\n", rc);

/* Commit transaction on connection 2 */
SQLEndTran(SQL_HANDLE_DBC, hdbc[1], SQL_COMMIT);

/* Pause to allow the existence of the tables to be verified. */
printf("Tables created, hit Return to continue\n");
getchar();

SQLSetConnection(hdbc[0]);
if (( rc = Drop_Tab() ) != 0)
    printf("Error dropping Table on 1st connection, RC=%ld\n", rc);
```

```

/* Commit transaction on connection 1 */
SQLEndTran(SQL_HANDLE_DBC, hdbc[0], SQL_COMMIT);

SQLSetConnection(hdbc[1]);
if (( rc = Drop_Tab() ) != 0)
    printf("Error dropping Table on 2nd connection, RC=%ld\n", rc);

/* Commit transaction on connection 2 */
SQLEndTran(SQL_HANDLE_DBC, hdbc[1], SQL_COMMIT);

printf("Tables dropped\n");
/***** End Processing Step *****/

/* ... */
/***** Embedded SQL Functions *****/
** This would normally be a separate file to avoid having to      *
** keep precompiling the embedded file in order to compile the DB2 CLI *
** section50                                                         *
*****/

EXEC SQL INCLUDE SQLCA;

int
Create_Tab( )
{
    EXEC SQL CREATE TABLE mixedup
        (ID INTEGER, NAME CHAR(10));

    return( SQLCODE);
}

int
Drop_Tab( )
{
    EXEC SQL DROP TABLE mixedup;

    return( SQLCODE);
}
/* ... */

```

Using 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 as defined by X/Open.

Note: ODBC defines short forms of vendor escape clauses that are not defined by X/Open.

Currently, escape clauses are used extensively by ODBC to define SQL extensions. DB2 ODBC translates the ODBC extensions into the correct DB2 syntax. The `SQLNativeSql()` function can be used to display the resulting syntax.

If an application is only going to access DB2 data sources, then there is no reason to use the escape clauses. If an application is going to access other data sources that offer the same support, but uses different syntax, then the escape clauses increase the portability of the application.

Escape clause syntax

The format of an X/Open SQL escape clause definition is:

```
--(*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 example, IBM).

product-identifier

Identifies an SQL product (for example, DB2).

These two parts make up the *SQL-escape-identification*.

Using ODBC defined SQL extensions

ODBC has used a vendor escape clause of:

```
--(* vendor(Microsoft), product(ODBC) extended SQL text*)--
```

to define the following SQL extensions (these extensions are not defined by X/Open):

- Extended date, time, timestamp data
- Outer join
- LIKE predicate
- Call stored procedure
- Extended scalar functions
 - Numeric functions
 - String functions
 - System functions

ODBC also defines a shorthand syntax for specifying these extensions:

```
{ extended SQL text }
```

X/Open does not support this shorthand syntax; however, it is widely used by ODBC applications.

ODBC date, time, timestamp data

The ODBC escape clauses for date, time, and timestamp data are:

```
--(*vendor(Microsoft),product(ODBC) d 'value'*)--  
--(*vendor(Microsoft),product(ODBC) t 'value'*)--  
--(*vendor(Microsoft),product(ODBC) ts 'value'*)--
```

d indicates *value* is a date in the *yyyy-mm-dd* format,

t indicates *value* is a time in the *hh:mm:ss* format

ts indicates *value* is a timestamp in the *yyyy-mm-dd hh:mm:ss.ffffff* format.

The shorthand syntax for date, time, and timestamp data is:

```
{d 'value'}  
{t 'value'}  
{ts 'value'}
```

For example, each of the following statements can be used to issue a query against the **EMPLOYEE** table:

```
SELECT * FROM EMPLOYEE  
WHERE HIREDATE=--(*vendor(Microsoft),product(ODBC) d '1994-03-29' *)--
```

```
SELECT * FROM EMPLOYEE WHERE HIREDATE={d '1994-03-29'}
```

DB2 ODBC translates either of the above statements to a DB2 format. `SQLNativeSql()` can be used to return the translated statement.

The ODBC escape clauses for date, time, and timestamp literals can be used in input parameters with a C data type of `SQL_C_CHAR`.

ODBC outer join syntax

The ODBC escape clause for outer join is:

```
--(*vendor(Microsoft),product(ODBC) oj outer join*)--
```

where *outer join* is:

```
table-name {LEFT | RIGHT | FULL} OUTER JOIN
           {table-name | outer-join}
           ON search-condition
```

Or alternatively, the ODBC shorthand syntax is:

```
{oj outer-join}
```

For example, DB2 ODBC translates the following two statements:

```
SELECT * FROM
--(*vendor(Microsoft),product(ODBC) oj
  T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3*)--
WHERE T1.C2>20

SELECT * FROM {oj T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3}
WHERE T1.C2>20
```

to IBM's format, which corresponds to the SQL92 outer join syntax.

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3 WHERE T1.C2>20
```

Note: Not all servers support outer join. To determine if the current server supports outer joins, call `SQLGetInfo()` with the `SQL_OUTER_JOIN` and `SQL_OJ_CAPABILITIES` options.

Like predicate escape clauses

In an SQL LIKE predicate, the metacharacter `%` matches zero or more of any character and the metacharacter `_` matches any one character. The `ESCAPE` clause allows the definition of patterns intended to match values that contain the actual percent and underscore characters by preceding them with an escape character. The escape clause ODBC uses to define the LIKE predicate escape character is:

```
--(*vendor(Microsoft),product(ODBC) escape 'escape-character'*)--
```

where *escape-character* is any character supported by the DB2 rules governing the use of the `ESCAPE` clause. The shorthand syntax for the LIKE predicate escape character is:

```
{escape 'escape-character'}
```

Applications that are not concerned about portability across different vendor DBMS products should pass the `ESCAPE` clause directly to the data source. To determine when LIKE predicate escape characters are supported by a particular data source, an application should call `SQLGetInfo()` with the `SQL_LIKE_ESCAPE_CLAUSE` information type.

Stored procedure CALL syntax

The ODBC escape clause for calling a stored procedure is:

```
--(*vendor(Microsoft),product(ODBC),  
  [?=]call procedure-name[[parameter][,[parameter]]...])*--
```

procedure-name

Specifies the name of a procedure stored at the data source.

parameter

Specifies a procedure parameter.

A procedure can have zero or more parameters. The short form syntax is:

```
{[?]=call procedure-name[[parameter][,[parameter]]...]}
```

(The square brackets ([]) indicate optional arguments.

ODBC specifies the optional parameter `?=` to represent the procedure's return value, which, if present, is stored in the location specified by the first parameter marker defined by `SQLBindParameter()`. DB2 ODBC returns the `SQLCODE` as the procedure's return value if `?=` is present in the escape clause. If `?=` is not present, then the application can retrieve the `SQLCA` by using the `SQLGetSQLCA()` function. Unlike ODBC, DB2 ODBC does not support literals as procedure arguments. Parameter markers must be used.

For more information about stored procedures, see "Using stored procedures" on page 417 or *DB2 Application Programming and SQL Guide*.

For example, DB2 ODBC translates the following two statements:

```
--(*vendor(Microsoft),product(ODBC) CALL NETB94(?,?,?)*--
```

```
{CALL NETB94(?,?,?)}
```

To an internal CALL statement format:

```
CALL NETB94(?, ?, ?)
```

ODBC scalar functions

Scalar functions such as string length, substring, or trim can be used on columns of a result sets and on columns that restrict rows of a result set. The ODBC escape clauses for scalar functions and its shorthand are:

```
--(*vendor(Microsoft),product(ODBC) fn scalar-function*--
```

or,

```
{fn scalar-function}
```

Where, *scalar-function* can be any function listed in Appendix B, "Extended scalar functions", on page 471.

For example, the ODBC driver translates both of the following statements:

```
SELECT --(*vendor(Microsoft),product(ODBC) fn CONCAT(FIRSTNAME, LASTNAME) *)--  
FROM EMPLOYEE
```

```
SELECT {fn CONCAT(FIRSTNAME, LASTNAME)} FROM EMPLOYEE
```

to:

```
SELECT FIRSTNAME CONCAT LASTNAME FROM EMPLOYEE
```

`SQLNativeSql()` can be called to obtain the translated SQL statement.

To determine which scalar functions are supported by the current server referenced by a specific connection handle, call `SQLGetInfo()` with the `SQL_NUMERIC_FUNCTIONS`, `SQL_STRING_FUNCTIONS`, `SQL_SYSTEM_FUNCTIONS`, and `SQL_TIMEDATE_FUNCTIONS` options.

Programming hints and tips

This section provides some hints and tips to help improve DB2 ODBC and ODBC application performance and portability.

Avoiding common initialization file problems

You can avoid two common problems when using the DB2 ODBC initialization file by ensuring that these contents 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.

Setting common connection options

The following connection options might need to be set (or considered) by DB2 ODBC applications.

SQL_AUTOCOMMIT

Generally this option should be set to `SQL_AUTOCOMMIT_OFF`, since each commit request can generate extra network flow. Only leave `SQL_AUTOCOMMIT` on if specifically needed.

Note: The default is `SQL_AUTOCOMMIT_ON`.

SQL_TXN_ISOLATION

This connection (and statement) option determines the isolation level at which the connection or statement operate. The isolation level determines the level of concurrency possible, and the level of locking required to execute the statement. Applications need to choose an isolation level that maximizes concurrency, yet ensures data consistency.

See Part 4 of *DB2 Application Programming and SQL Guide* for a complete discussion of isolation levels and their effect.

Setting common statement options

The following statement options might need to be set by DB2 ODBC applications.

SQL_MAX_ROWS

Setting this option limits the number of rows returned to the application. This can be used to avoid an application from being overwhelmed with a very large result set generated inadvertently, especially for applications on clients with limited memory resources.

Note: The full result set is still generated at the server. DB2 ODBC only fetches up to SQL_MAX_ROWS rows.

SQL_CURSOR_HOLD

This statement option determines if the cursor for this statement is defined with the equivalent of the CURSOR WITH HOLD clause.

Resources associated with statement handles can be better utilized by DB2 ODBC if the statements that do not require CURSOR WITH HOLD are set to SQL_CURSOR_HOLD_OFF.

Note: Many ODBC applications expect a default behavior where the cursor position is maintained after a commit.

SQL_STMTTXN_ISOLATION

DB2 ODBC allows the isolation level to be set at the statement level (however, it is best to set the isolation level at the connection level). The isolation level determines the level of concurrency possible, and the level of locking required to execute the statement.

Resources associated with statement handles can be better utilized by DB2 ODBC if statements are set to the required isolation level, rather than leaving all statements at the default isolation level. This should only be attempted with a thorough understanding of the locking and isolation levels of the connected DBMS. See *DB2 SQL Reference* for a complete discussion of isolation levels and their effect.

Applications should use the minimum isolation level possible to maximize concurrency.

Using SQLSetColAttributes to reduce network flow

Each time a query statement is prepared or executed directly, DB2 ODBC retrieves information about the SQL data type, and its size from the data source. If the application *knows* this information ahead of time, SQLSetColAttributes() can be used to provide DB2 ODBC with the information. This can significantly reduce the network flow from remote data sources if the result set coming back contains a very large number (hundreds) of columns.

Note: The application must provide DB2 ODBC with exact result descriptor information for ALL columns; otherwise, an error occurs when the data is fetched.

Queries that generate result sets that contain a large number of columns, but relatively small number of rows, have the most to gain from using SQLSetColAttributes().

Comparing binding and SQLGetData

Generally it is more efficient to bind application variables to result sets than to use SQLGetData(). Use SQLGetData() when the data value is large variable-length data that:

- Must be received in pieces, or
- Might not need to be retrieved (dependent on another application action.)

Increasing transfer efficiency

The efficiency of transferring character data between bound application variables and DB2 ODBC can be increased if the *pcbValue* and *rgbValue* arguments are contiguous in memory. (This allows DB2 ODBC to fetch both values with one copy operation.)

For example:

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

Limiting use of catalog functions

In general, try to limit the number of times the catalog functions are called, and limit the number of rows returned.

The number of catalog function calls can be reduced by calling the function once, and storing the information at the application.

The number of rows returned can be limited by specifying a:

- Schema name or pattern for all catalog functions
- Table name or pattern for all catalog functions other than `SQLTables`
- Column name or pattern for catalog functions that return detailed column information.

Remember, although an application can be developed and tested against a data source with hundreds of tables, it can be run against a database with thousands of tables. Plan ahead.

Close any open cursors (call `SQLCloseCursor()`) for statement handles used for catalog queries to release any locks against the catalog tables. Outstanding locks on the catalog tables can prevent `CREATE`, `DROP` or `ALTER` statements from executing.

Using column names of function generated result sets

The column names of the result sets generated by catalog and information functions can change as the X/Open and ISO standards evolve. The *position* of the columns, however, does not change.

Any application dependency should be based on the column position (*icol* parameter) and not the name.

Making use of dynamic SQL statement caching

To make use of *dynamic caching* (when the server caches a prepared version of a dynamic SQL statement), the application must use the same statement handle for the same SQL statement.

For example, if an application routinely uses a set of 10 SQL statements, 10 statement handles should be allocated and associated with each of those statements. Do not free the statement handle while the statement can still be executed. (The transaction can still be rolled back or committed without affecting any of the prepared statements). The application continues to prepare and execute the statements in a normal manner. The DB2 server determines if the prepare is actually needed.

To reduce function call overhead, the statement can be prepared once, and executed repeatedly throughout the application.

Optimizing insertion and retrieval of data

The methods described in “Using arrays to input parameter values” on page 403 and “Retrieving a result set into an array” on page 406 optimize the network flow.

Use these methods as much as possible.

Optimizing for large object data

Use LOB data types and the supporting functions for long strings whenever possible. Unlike LONG VARCHAR, LONG VARBINARY, and LONG VARCHARIC data types, LOB data values can use LOB locators and functions such as SQLGetPosition() and SQLGetSubString() to manipulate large data values at the server.

Using SQLDriverConnect instead of SQLConnect

Using SQLDriverConnect() gives an application the flexibility to override any or all of the initialization keyword values specified for the target data source.

Turning off statement scanning

DB2 ODBC by default, scans each SQL statement searching for vendor escape clause sequences.

If the application does not generate SQL statements that contain vendor escape clause sequences (“Using vendor escape clauses” on page 448), then the SQL_NO_SCAN statement option should be set to SQL_NOSCAN_ON at the connection level so that DB2 ODBC does not perform a scan for vendor escape clauses.

Casting distinct types

If a parameter marker is used in the predicate of a query statement, and the parameter is a distinct type, the statement must use a CAST function to cast either the parameter marker or the distinct type.

For example, assume the following distinct type and table are defined:

```
CREATE DISTINCT TYPE CNUM AS INTEGER WITH COMPARISONS
CREATE TABLE CUSTOMER (
    Cust_Num      CNUM NOT NULL,
    First_Name    CHAR(30) NOT NULL,
    Last_Name     CHAR(30) NOT NULL,
    Phone_Num     CHAR(20) WITH DEFAULT,
    PRIMARY KEY  (Cust_Num) )
```

The following statement fails because the parameter marker cannot be type CNUM and the comparison fails due to incompatible types:

```
SELECT first_name, last_name, phone_num FROM customer
       where cust_num = ?
```

Alternatively the parameter marker can be cast to INTEGER. The server can then apply the INTEGER to CNUM conversion:

```
SELECT first_name, last_name, phone_num FROM customer
       where cust_num = cast( ? as integer )
```

See *DB2 SQL Reference* for more information about parameter markers (PREPARE statement) and casting (CAST function).

Chapter 7. Problem diagnosis

This section provides guidelines for working with the DB2 ODBC traces and information about general diagnosis, debugging, and abends. You can obtain traces for DB2 ODBC applications and diagnostics and DB2 ODBC stored procedures.

Tracing

DB2 ODBC provides two traces that differ in purpose:

- An application trace intended for debugging user applications, described in “Application trace”.
- A service trace for problem diagnosis, described in “Diagnostic trace” on page 459.

Application trace

The DB2 ODBC application trace is enabled using the APPLTRACE and APPLTRACEFILENAME keywords in the DB2 ODBC initialization file.

The APPLTRACE keyword is intended for customer application debugging. This trace records data information at the DB2 ODBC API interface; it is specifically designed to trace ODBC API calls. The trace is written to the file specified on the APPLTRACEFILENAME keyword. We strongly recommend that you use this trace

Specifying the trace file name

You can use a JCL DD card format or an OS/390 UNIX HFS file name format to specify the APPLTRACEFILENAME keyword setting. The primary use of the JCL DD card format is write to an MVS preallocated sequential data set. You can also specify OS/390 UNIX HFS files on a DD statement. The OS/390 UNIX HFS file name format is used strictly for writing to HFS files.

JCL DD card format: The JCL DD card format is APPLTRACEFILENAME="DD:ddname". The ddname value is the name of the DD card specified in your job or TSO logon procedure.

Examples: Assume the keyword setting is APPLTRACEFILENAME="DD:APPLDD". You can use the following JCL DD statement examples in your job or TSO logon procedure to specify the MVS trace data set.

Example 1: Write to preallocated MVS sequential data set USER01.MYTRACE.

```
//APPLDD DD DISP=SHR,DSN=USER01.MYTRACE
```

Example 2: Write to preallocated OS/390 UNIX HFS file MYTRACE in directory /usr/db2.

```
//APPLDD DD PATH='/usr/db2/MYTRACE'
```

Example 3: Allocate OS/390 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)
```

OS/390 UNIX HFS file name format: The OS/390 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.

Examples: The following examples use the APPLTRACEFILENAME keyword to specify an OS/390 UNIX HFS trace file.

Example 1: Create and write to HFS file named APPLTRC1 in the fully qualified directory /usr/db2.

```
APPLTRACEFILENAME=/usr/db2/APPLTRC1
```

Example 2: Create and write to HFS file named APPLTRC1 in the current working directory of the application.

```
APPLTRACEFILENAME=./APPLTRC1
```

Example 3: Create and write to HFS file named APPLTRC1 in the parent directory of the current working directory.

```
APPLTRACEFILENAME=../APPLTRC1
```

Application trace output

The following example of application trace output shows how DB2 ODBC follows the APIs invoked, indicates values used, data pointers, etc. Errors are also indicated.

```
SQLAllocHandle( fHandleType=SQL_HANDLE_ENV, hInput=0, phOutput=&6b7e77c )
SQLAllocHandle( phOutput=1 )
----> SQL_SUCCESS

SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=1, phOutput=&6b7e778 )
SQLAllocHandle( phOutput=1 )
----> SQL_SUCCESS

SQLConnect( hDbc=1, szDSN=NULL Pointer, cbDSN=0, szUID=NULL Pointer, cbUID=0,
szAuthStr=NULL Pointer, cbAuthStr=0 )
SQLConnect( )
----> SQL_SUCCESS

SQLAllocHandle( fHandleType=SQL_HANDLE_STMT, hInput=1, phOutput=&6b7e774 )
SQLAllocHandle( phOutput=1 )
----> SQL_SUCCESS

SQLExecDirect( hStmt=1, pszSqlStr="SELECT NAME FROM SYSIBM.SYSPLAN", cbSqlStr=-3 )
SQLExecDirect( )
----> SQL_SUCCESS

SQLFetch( hStmt=1 )
SQLFetch( )
----> SQL_SUCCESS

SQLEndTran( fHandleType=SQL_HANDLE_DBC, fHandle=1, fType=SQL_COMMIT )
SQLEndTran( )
----> SQL_SUCCESS

SQLFreeHandle( fHandleType=SQL_HANDLE_STMT, hHandle=1 )
SQLFreeHandle( )
----> SQL_SUCCESS

SQLDisconnect( hDbc=1 )
SQLDisconnect( )
----> SQL_SUCCESS

SQLFreeHandle( fHandleType=SQL_HANDLE_DBC, hHandle=1 )
SQLFreeHandle( )
----> SQL_SUCCESS
```

```
SQLFreeHandle( fHandleType=SQL_HANDLE_ENV, hHandle=1 )
SQLFreeHandle()
----> SQL_SUCCESS
```

For more information about how to specify the APPLTRACE and APPLTRACEFILENAME keywords, see “DB2 ODBC initialization file” on page 52.

Diagnostic trace

The DB2 ODBC diagnostic trace captures information to use in DB2 ODBC problem determination. The trace is intended for use under the direction of the IBM Support Center; it is not intended to assist in debugging user written DB2 ODBC applications. You can view this trace to obtain information about the general flow of an application, such as commit information. However, this trace is intended for IBM service information only and is therefore subject to change.

You can activate the diagnostic trace by either the DSNAOTRC command or the DIAGTRACE keyword in the DB2 ODBC initialization file.

If you activate the diagnostic trace using the DIAGTRACE keyword in the initialization file, you must also allocate a DSNAOTRC DD statement in your job or TSO logon procedure. You can use one of the following methods to allocate a DSNAOTRC DD statement:

- Specify a DSNAOTRC DD JCL statement in your job or TSO logon procedure
- Use the TSO/E ALLOCATE command
- Use dynamic allocation in your ODBC application

Specifying the diagnostic trace file

The diagnostic trace data can be written to an MVS sequential data set or an OS/390 UNIX HFS file.

An MVS 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 OS/390 UNIX 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 using the trace formatting program.

Examples: The following JCL examples use a DSNAOTRC JCL DD card to specify the diagnostic trace file.

Example 1: Write to preallocated MVS sequential data set USER01.DIAGTRC.

```
//DSNAOTRC DD      DISP=SHR,DSN=USER01.DIAGTRC
```

Example 2: Write to preallocated OS/390 UNIX HFS file DIAGTRC in directory /usr/db2.

```
//DSNAOTRC      DD      PATH='/usr/db2/DIAGTRC'
```

Example 3: Allocate OS/390 UNIX HFS file DIAGTRC in directory /usr/db2 specifying permission for the file owner to read from (SIRUSR) and write to (SIWUSR) the trace file.

```
//DSNAOTRC      DD  PATH='/usr/db2/DIAGTRC',  
                PATHOPTS=(ORDWR,OCREAT,OTRUNC),  
                PATHMODE=(SIRUSR,SIWUSR)
```

For more information about the DIAGTRACE keyword, see “DB2 ODBC initialization file” on page 52.

Using the diagnostic trace command: DSNAOTRC

You can use the DSNAOTRC command to:

- 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 an MVS data set or OS/390 UNIX HFS file.
- Format the DB2 ODBC diagnostic trace.

Special OS/390 UNIX considerations: You can issue the DSNAOTRC command from the OS/390 UNIX shell command line to activate the diagnostic trace prior to executing an ODBC application. Under the direction of IBM support only, you must store the dsnaotrc program load module in an OS/390 UNIX HFS file.

Use the TSO/E command, OPUTX, to store the dsnaotrc load module in an HFS file. The following example uses the OPUTX command to store load module dsnaotrc from MVS partitioned data set DB2A.DSNLOAD to HFS file DSNAOTRC in directory /usr/db2:

```
OPUTX 'DB2A.DSNLOAD(DSNAOTRC)' /usr/db2/dsnaotrc
```

After storing the dsnaotrc program module in an HFS file, follow these steps at the OS/390 UNIX shell to activate, dump, and format the diagnostic trace:

1. Enable the shared address space environment variable for the OS/390 UNIX shell. Issue the following export statement at the command line or specify it in your \$HOME/.profile:

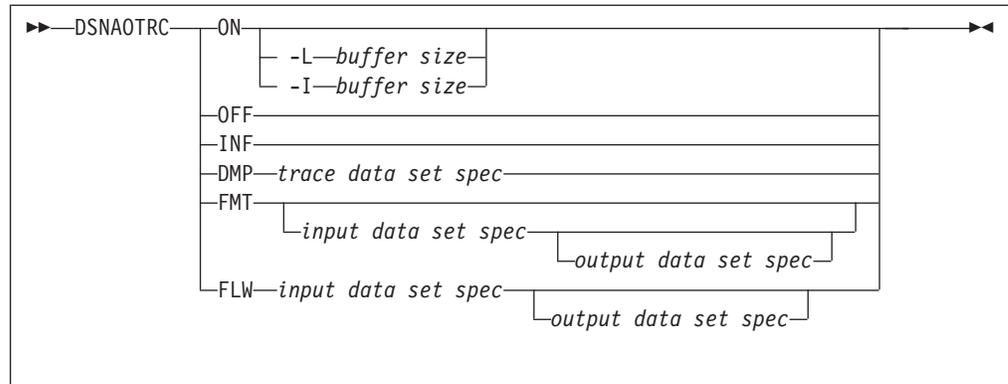
```
export _BPX_SHAREAS=YES
```

Setting this environment variable allows the OMVS command and the OS/390 UNIX shell to run in the same TSO address space.

2. Go to the directory that contains the dsnaotrc module.
3. Verify that execute permission is established for the dsnaotrc load module. If execute permission was not granted, use the chmod command to set execute permission for the dsnaotrc load module.
4. Issue dsnaotrc on. The options for activating the diagnostic trace are optional.
5. Execute the ODBC application.
6. Issue dsnaotrc dmp "raw_trace_file". The raw_trace_file value is the name of the output file to which DB2 writes the raw diagnostic trace data.
7. Issue dsnaotrc off to deactivate the diagnostic trace.
8. Issue dsnaotrc fmt "raw_trace_file" "fmt_trace_file" to format the raw trace data records from input file "raw_trace_file" to output file "fmt_trace_file".

After successfully formatting the diagnostic trace data, delete the dsnaotrc program module from your OS/390 UNIX directory. Do not attempt to maintain a private copy of the dsnaotrc program module in your HFS directory.

Syntax:



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 MVS data set or OS/390 UNIX HFS file to which DB2 writes the raw DB2 ODBC diagnostic trace data. The data set specification can be either an MVS data set name, an OS/390 UNIX HFS file name, or a currently allocated JCL DD card name.

FMT

Generate a formatted detail report of the DB2 ODBC diagnostic trace contents.

FLW

Generate a formatted flow report of the DB2 ODBC diagnostic trace contents.

input data set spec

The data set that contains the raw DB2 ODBC diagnostic trace data to be formatted. This is the data set that was generated as the result of a DSNAOTRC DMP command or due to the DSNAOTRC DD card if the trace was started by using the DIAGTRACE initialization keyword. The data set specification can be either an MVS data set name, an OS/390 UNIX HFS file name, or a currently allocated JCL DD card name. If this parameter is not specified, then DSNAOTRC 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 an MVS data set name, an OS/390 UNIX HFS file name, or a currently allocated JCL DD card name. If you specify an MVS data set or OS/390 UNIX HFS file that does not exist, DB2 allocates it dynamically. If this parameter is not specified, the output is written to standard output ("STDOUT").

Examples: The following examples show how to code the data set specifications.

- Trace data set specification:

Example 1: Currently allocated JCL DD card name TRACEDD.

```
DSNAOTRC DMP DD:TRACEDD
```

Example 2: MVS sequential data set USER01.DIAGTRC.

```
DSNAOTRC DMP "USER01.DIAGTRC"
```

Example 3: OS/390 UNIX HFS file named DIAGTRC in directory /usr/db2:

```
DSNAOTRC DMP "/usr/db2/DIAGTRC"
```

- Input data set specification:

Example 1: Currently allocated JCL DD card name INPDD.

```
DSNAOTRC FLW DD:INPDD output-dataset-spec
```

Example 2: MVS sequential data set USER01.DIAGTRC.

```
DSNAOTRC FLW "USER01.DIAGTRC" output-dataset-spec
```

Example 3: OS/390 UNIX HFS file DIAGTRC in directory /usr/db2.

```
DSNAOTRC FLW "/usr/db2/DIAGTRC" output-dataset-spec
```

- Output data set specification:

Example 1: Currently allocated JCL DD card name OUTPDD.

```
DSNAOTRC FLW input-dataset-spec DD:OUTPDD
```

Example 2: MVS sequential data set USER01.TRCFLOW.

```
DSNAOTRC FLW input-dataset-spec "USER01.TRCFLOW"
```

Example 3: OS/390 UNIX HFS file TRCFLOW in directory /usr/db2.

```
DSNAOTRC FLW input-dataset-spec "/usr/db2/TRCFLOW"
```

Stored procedure trace

This section describes the steps required to obtain an application trace or a diagnostic trace of a DB2 ODBC stored procedure. DB2 ODBC stored procedures run in either a DB2-established stored procedures address space or a WLM-established address space. Both the main application that calls the stored procedure (client application), and the stored procedure itself, can be either a DB2 ODBC application or a standard DB2 precompiled application.

If the client application and the stored procedure are DB2 ODBC application programs, you can trace:

- A client application only
- A stored procedure only
- Both the client application and stored procedure

More than one address spaces can not share write access to a single data set. Therefore, you must use the appropriate JCL DD statements to allocate a unique trace data set for each stored procedures address space that uses the DB2 ODBC application trace or diagnostic trace.

Tracing a client application

This section explains how to obtain an application trace and a diagnostic trace for a client application.

Application trace: Follow these steps to obtain an application trace.

1. Set APPLTRACE=1 and APPLTRACEFILENAME="DD:DDNAME" in the common section of the DB2 ODBC initialization file as follows:

```
[COMMON]
APPLTRACE=1
APPLTRACEFILENAME="DD:APPLTRC"
```

DDNAME is the name of an OS/390 JCL DD statement specified in the JCL for the application job or your TSO logon procedure.

2. Specify an OS/390 JCL DD statement in the JCL for the application job or your TSO logon procedure. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes RECFM=VBA, LRECL=137, an OS/390 UNIX HFS file to contain the client application trace, as shown in the following examples:

```
//APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC
//APPLTRC DD PATH='/u/cli/appltrc'
```

Diagnostic trace: When tracing only the client application, you can activate the diagnostic trace by using the DIAGTRACE keyword in the DB2 ODBC initialization file or the DSNAOTRC command. See "Diagnostic trace" on page 459 for information about obtaining a diagnostic trace of the client application.

Tracing a stored procedure

This section explains how to obtain an application trace and a diagnostic trace for a stored procedure.

Application trace: Follow these steps to obtain an application trace.

1. Set APPLTRACE=1 and APPLTRACEFILENAME="DD:DDNAME" in the common section of the DB2 ODBC initialization file as follows:

```
[COMMON]
APPLTRACE=1
APPLTRACEFILENAME="DD:APPLTRC"
```

DDNAME is the name of an OS/390 JCL DD statement specified in the JCL for the stored procedures address space.

2. Specify an OS/390 JCL DD statement in the JCL for the stored procedures address space. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes RECFM=VBA, LRECL=137 or OS/390 UNIX HFS file to contain the client application trace, as shown in the following examples:

```
//APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC
//APPLTRC DD PATH='/u/cli/appltrc'
```

Diagnostic trace: Follow these steps to obtain a diagnostic trace.

1. Set DIAGTRACE=1, DIAGTRACE_BUFFER_SIZE=nnnnnnn, and DIAGTRACE_NO_WRAP=0|1 in the common section of the DB2 ODBC initialization file. For example:

```
[COMMON]
DIAGTRACE=1
DIAGTRACE_BUFFER_SIZE=2000000
DIAGTRACE_NO_WRAP=1
```

nnnnnnn is the number of bytes to allocate for the diagnostic trace buffer.

2. Specify an OS/390 DSNAOINI JCL DD statement in the JCL for the stored procedures 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 an OS/390 DSNAOTRC JCL DD statement in the JCL for the stored procedures space. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes RECFM=FB, LRECL=80, or an OS/390 UNIX HFS file to contain the unformatted diagnostic data, as shown in the following examples:


```
//DSNAOTRC DD DISP=SHR,DSN=CLI.DIAGTRC
//DSNAOTRC DD PATH='/u/cli/diagtrc'
```
4. Execute the client application that calls the stored procedure.
5. After the DB2 ODBC stored procedure executes, stop the stored procedures address space.
 - For DB2-established address spaces, use the DB2 command, STOP PROCEDURE.
 - For WLM-established address spaces operating in WLM goal mode, use the MVS command, "VARY WLM,APPLENV=name,QUIESCE". name is the WLM application environment name.
 - For WLM-established address spaces operating in WLM compatibility mode, use the MVS command, "CANCEL address-space-name". address-space-name is the name of the WLM-established address space.
6. You can submit either the formatted or unformatted diagnostic trace data to the IBM Support Center. To format the raw trace data at your site, run the DSNAOTRC FMT or DSNAOTRC FLW command against the diagnostic trace data set.

Tracing both a client application and a stored procedure

This section explains how to obtain an application trace and a diagnostic trace for both a client application and a stored procedure.

Application trace: Follow these steps to obtain an application trace.

1. Set APPLTRACE=1 and APPLTRACEFILENAME="DD:DDNAME" in the common section of the DB2 ODBC initialization file as follows:

```
[COMMON]
APPLTRACE=1
APPLTRACEFILENAME="DD:APPLTRC"
```

DDNAME is the name of an OS/390 JCL DD statement specified in both the JCL for the client application job and the stored procedures address space.

2. Specify an OS/390 JCL DD statement in the JCL for the client application. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes RECFM=VBA, LRECL=137, or an OS/390 UNIX 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 an OS/390 JCL DD statement in the JCL for the stored procedures address space. The DD statement references a pre-allocated OS/390 sequential data set, or an OS/390 UNIX HFS file to contain the stored procedure application trace, as shown in the following examples:

```
//APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC.SPROC
//APPLTRC DD PATH='/u/cli/appltrc.sproc'
```

You must allocate a separate trace data set or HFS file for the stored procedure. Do not attempt to write to the same application trace data set or HFS file used for the client application.

Diagnostic trace: Follow these steps to obtain a diagnostic trace.

1. Set DIAGTRACE=1, DIAGTRACE_BUFFER_SIZE=nnnnnnn, and DIAGTRACE_NO_WRAP=0|1 in the common section of the DB2 ODBC initialization file. For example:

```
[COMMON]
DIAGTRACE=1
DIAGTRACE_BUFFER_SIZE=2000000
DIAGTRACE_NO_WRAP=1
```

nnnnnnn is the number of bytes to allocate for the diagnostic trace buffer.

2. Specify an OS/390 DSNAOINI JCL DD statement in the JCL for the stored procedures 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 an OS/390 DSNAOTRC JCL DD statement in JCL for the client application job. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes RECFM=FB, LRECL=80, or an OS/390 UNIX 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 an OS/390 DSNAOTRC JCL DD statement in the JCL for the stored procedures address space. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes RECFM=FB, LRECL=80, or an OS/390 UNIX HFS file to contain the stored procedure's unformatted diagnostic data, as shown in the following examples:

```
//DSNAOTRC DD DISP=SHR,DSN=CLI.DIAGTRC.SPROC
//DSNAOTRC DD PATH='/u/cli/diagtrc.sproc'
```

5. Execute the client application that calls the stored procedure.
6. After the DB2 ODBC stored procedure executes, stop the stored procedures address space.
 - For DB2-established address spaces, use the DB2 command, STOP PROCEDURE.
 - For WLM-established address spaces operating in WLM goal mode, use the MVS command, "VARY WLM,APPLENV=name,QUIESCE". name is the WLM application environment name.
 - For WLM-established address spaces operating in WLM compatibility mode, use the MVS command, "CANCEL address-space-name". address-space-name is the name of the WLM-established address space.

7. You can submit either the formatted or unformatted diagnostic trace data to the IBM Support Center. To format the raw trace data at your site, run the DSNAOTRC FMT or DSNAOTRC FLW command against the client application's diagnostic trace data set and the stored procedure's diagnostic trace data set.

Debugging

You can debug DB2 for OS/390 and z/OS ODBC applications debug tool shipped with your the C or C++ language compiler. For detailed instructions on debugging DB2 stored procedures, including DB2 ODBC stored procedures, see Part 6 of *DB2 Application Programming and SQL Guide*.

Abnormal termination

Language Environment reports abends since DB2 ODBC runs under Language Environment. Typically, Language Environment reports the type of abend that occurs and the function that is active in the address space at the time of the abend.

DB2 ODBC has no facility for abend recovery. When an abend occurs, DB2 ODBC terminates. DBMSs follow the normal recovery process for any outstanding DB2 unit of work.

"CEE" is the prefix for all Language Environment messages. If the prefix of the active function is "CLI", then DB2 ODBC had control during the abend which indicates that this can be a DB2 ODBC, a DB2, or a user error.

The following example shows an abend:

```
CEE3250C The system or user abend S04E R=00000000 was issued.  
  From entry point CLI_mvscallProcedure(CLI_CONNECTINFO*,...  
  +091A2376 at address 091A2376...
```

In this message, you can determine what caused the abend as follows:

- "CEE" indicates that Language Environment is reporting the abend.
- The entry point shows that DB2 ODBC is the active module.
- Abend code "S04E" means that this is a DB2 system abend.

For further information on debugging, see *OS/390 Language Environment for OS/390 & VM Debugging Guide*. For further information on the DB2 recovery process, see Part 4 (Volume 1) of *DB2 Administration Guide*.

Internal error code

DB2 ODBC provides an internal error code for ODBC diagnosis that is intended for use under the guidance of IBM service. This unique error location, ERRLOC, is a good tool for APAR searches. The following example of a failed `SQLAllocHandle()` (with *HandleType* set to `SQL_HANDLE_DBC`) shows an error location:

```
DB2 ODBC Sample SQLError Information  
DB2 ODBC Sample SQLSTATE           : 58004  
DB2 ODBC Sample Native Error Code  : -99999  
DB2 ODBC Sample Error message text:  
  {DB2 for OS/390}{ODBC Driver} SQLSTATE=58004  ERRLOC=2:170:4;  
  RRS "IDENTIFY" failed using DB2 system:V71A,  
  RC=08 and REASON=00F30091
```

Appendix A. DB2 ODBC and ODBC

This appendix explains the differences between DB2 ODBC and ODBC in the following areas:

- “DB2 ODBC and ODBC drivers”
- “ODBC APIs and data types” on page 468
- “Isolation levels” on page 470

For a complete list of functions that DB2 ODBC and ODBC support, see Table 11 on page 68.

DB2 ODBC and ODBC drivers

This section discusses the support provided by the ODBC driver, and how it differs from DB2 ODBC.

Figure 22 below compares DB2 ODBC and the DB2 ODBC driver.

1. An ODBC driver under the ODBC driver manager
2. DB2 ODBC, callable interface designed for DB2 specific applications.

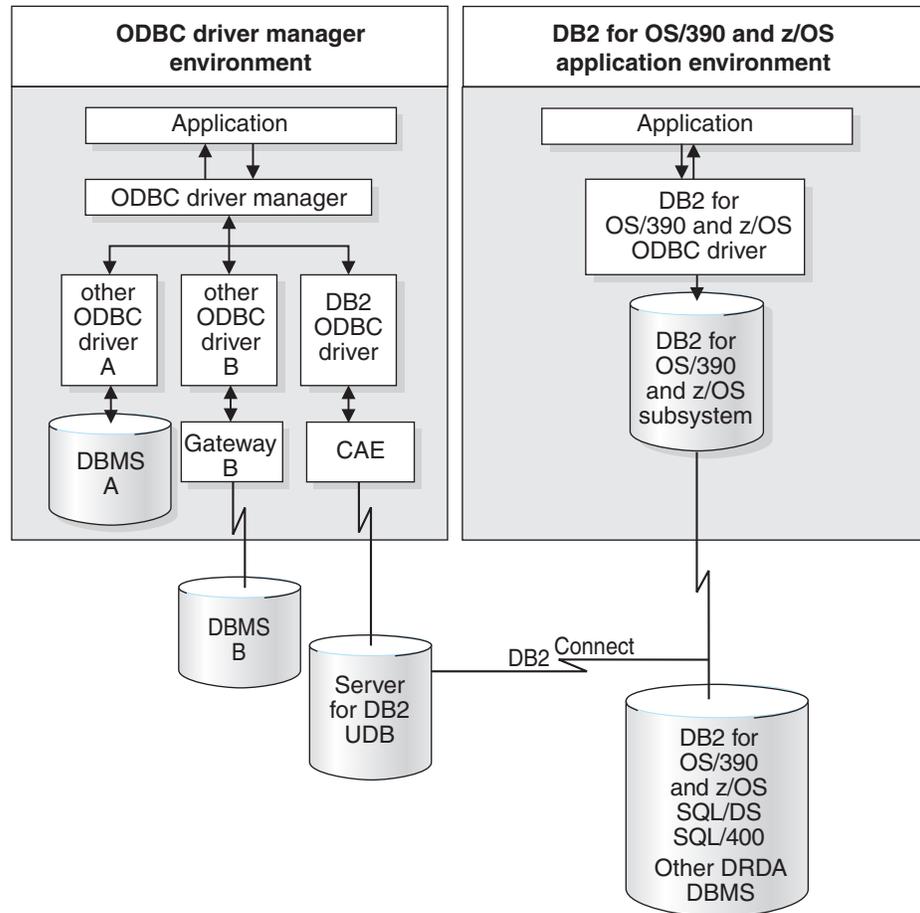


Figure 22. DB2 ODBC and ODBC.

In an ODBC environment, the driver manager provides the interface to the application. It also dynamically loads the necessary *driver* for the database server to

which the application connects. It is the driver that implements the ODBC function set, with the exception of some extended functions 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 provided by the ODBC driver.

DB2 ODBC applications interface directly with the ODBC driver which executes within the application address space. Applications do not interface with a driver manager. The capabilities provided to the application are a subset of the Microsoft ODBC Version 2 specifications.

ODBC APIs and data types

Table 171 summarizes the ODBC Version 2 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 11 on page 68 provides a complete list of functions supported by DB2 ODBC and ODBC 2.0.

Table 171. DB2 ODBC support

ODBC features	DB2 ODBC
Core level functions	All
Level 1 functions	All
Level 2 functions	All, except for: <ul style="list-style-type: none"> • SQLBrowseConnect() • SQLDrivers() • SQLSetPos() • SQLSetScrollOptions()
Additional DB2 ODBC functions	<ul style="list-style-type: none"> • SQLCancel() • SQLSetConnection() • SQLGetEnvAttr() • SQLSetEnvAttr() • SQLSetColAttributes() • SQLGetSQLCA()
Minimum SQL data types	<ul style="list-style-type: none"> • SQL_CHAR • SQL_LONGVARCHAR • SQL_VARCHAR
Core SQL data types	<ul style="list-style-type: none"> • SQL_DECIMAL • SQL_NUMERIC • SQL_SMALLINT • SQL_INTEGER • SQL_REAL • SQL_FLOAT • SQL_DOUBLE

Table 171. DB2 ODBC support (continued)

ODBC features	DB2 ODBC
Extended SQL data types	<ul style="list-style-type: none"> • SQL_BIT • SQL_TINYINT • SQL_BIGINT (NOT SUPPORTED) • SQL_BINARY • SQL_BLOB • SQL_BLOB_LOCATOR • SQL_CLOB • SQL_CLOB_LOCATOR • SQL_DBCLOB • SQL_DBCLOB_LOCATOR • SQL_LONGVARBINARY • SQL_ROWID • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_VARBINARY
ODBC Version 3 SQL data types	<ul style="list-style-type: none"> • SQL_GRAPHIC • SQL_LONGVARGRAPHIC • SQL_VARGRAPHIC
Core C data types	<ul style="list-style-type: none"> • SQL_C_CHAR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG(SLONG, ULONG) • SQL_C_SHORT (SSHORT, USHORT)
Extended C data types	<ul style="list-style-type: none"> • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TINYINT
ODBC Version 3 C data types	<ul style="list-style-type: none"> • SQL_C_DBCHAR
Return codes	<ul style="list-style-type: none"> • SQL_SUCCESS • SQL_SUCCESS_WITH_INFO • SQL_NEED_DATA • SQL_NO_DATA_FOUND • SQL_ERROR • SQL_INVALID_HANDLE
SQLSTATEs	Mapped to X/Open SQLSTATEs with additional IBM SQLSTATEs
Multiple connections per application	Supported but type 1 connections, SQL_CONNECTTYPE = SQL_CONCURRENT_TRANS. Must be on a transaction boundary prior to an SQLConnect or SQLSetConnection.

For more information about ODBC, see *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*.

Isolation levels

The following table maps IBM RDBMs isolation levels to ODBC transaction isolation levels. The `SQLGetInfo()` function, indicates which isolation levels are available.

Table 172. Isolation levels under ODBC

IBM isolation level	ODBC isolation level
Cursor stability	SQL_TXN_READ_COMMITTED
Repeatable read	SQL_TXN_SERIALIZABLE_READ
Read stability	SQL_TXN_REPEATABLE_READ
Uncommitted read	SQL_TXN_READ_UNCOMMITTED
No commit	(no equivalent in ODBC)

Note: `SQLSetConnectAttr()` and `SQLSetStmtAttr` return `SQL_ERROR` with an `SQLSTATE` of `S1009` if you try to set an unsupported isolation level.

Appendix B. Extended scalar functions

The following functions are defined by ODBC using vendor escape clauses. Each function can be called using the escape clause syntax, or calling the equivalent DB2 function.

These functions are presented in the following categories:

- “String functions”
- “Date and time functions” on page 472
- “System functions” on page 472

For more information about vendor escape clauses, see “ODBC scalar functions” on page 451.

All errors detected by the following functions, when connected to a DB2 UDB Version 2 server, return SQLSTATE 38552. The text portion of the message is of the form SYSFUN:*nn* where *nn* is one of the following reason codes:

01	Numeric value out of range
02	Division by zero
03	Arithmetic overflow or underflow
04	Invalid date format
05	Invalid time format
06	Invalid timestamp format
07	Invalid character representation of a timestamp duration
08	Invalid interval type (must be one of 1, 2, 4, 8, 16, 32, 64, 128, 256)
09	String too long
10	Length or position in string function out of range
11	Invalid character representation of a floating point number

String functions

The string functions in this section are supported by DB2 ODBC and defined by ODBC using vendor escape clauses.

Note:

- Character string literals used as arguments to scalar functions must be bounded by single quotes.
- Arguments denoted as *string_exp* can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, 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.

String functions

LEFT(*string_exp*,*count*)

Returns the leftmost *count* of characters of *string_exp*.

LENGTH(*string_exp*)

Returns the number of characters in *string_exp*, excluding trailing blanks and the string termination character.

REPEAT(*string_exp*, *count*)

Returns a character string composed of *string_exp* repeated *count* times.

RIGHT(*string_exp*, *count*)

Returns the rightmost count of characters of *string_exp*.

SUBSTRING(*string_exp*, *start*, *length*)

Returns a character string that is derived from *string_exp* beginning at the character position specified by *start* for *length* characters.

Date and time functions

The date and time functions in this section are supported by DB2 ODBC and defined by ODBC using vendor escape clauses.

Note:

- Arguments denoted as *timestamp_exp* can be the name of a column, the result of another scalar function, or a time, date, or timestamp literal.
- Arguments denoted as *date_exp* can be the name of a column, the result of another scalar function, or a date or timestamp literal, where the underlying data type can be character based, or date or timestamp based.
- Arguments denoted as *time_exp* can be the name of a column, the result of another scalar function, or a time or timestamp literal, where the underlying data types can be character based, or time or timestamp based.

CURDATE()

Returns the current date as a date value.

CURTIME()

Returns the current local time as a time value.

DAYOFMONTH (*date_exp*)

Returns the day of the month in *date_exp* as an integer value in the range of 1-31.

HOUR(*time_exp*)

Returns the hour in *time_exp* as an integer value in the range of 0-23.

MINUTE(*time_exp*)

Returns the minute in *time_exp* as integer value in the range of 0-59.

MONTH(*date_exp*)

Returns the month in *date_exp* as an integer value in the range of 1-12.

NOW()

Returns the current date and time as a timestamp value.

SECOND(*time_exp*)

Returns the second in *time_exp* as an integer value in the range of 0-59.

System functions

The system functions in this section are supported by DB2 ODBC and defined by ODBC using vendor escape clauses.

- Arguments denoted as *exp* can be the name of a column, the result of another scalar function, or a literal.
- Arguments denoted as *value* can be a literal constant.

DATABASE()

Returns the name of the database corresponding to the connection handle (*hdbc*). (The name of the database is also available using `SQLGetInfo()` by specifying the information type `SQL_DATABASE_NAME`.)

IFNULL(*exp*, *value*)

If *exp* is null, *value* is returned. If *exp* is not null, *exp* is returned. The possible data types of *value* must be compatible with the data type of *exp*.

USER()

Returns the user's authorization name. (The user's authorization name is also available using `SQLGetInfo()` by specifying the information type `SQL_USER_NAME`.)

System functions

Appendix C. SQLSTATE cross reference

This table is a cross-reference of all the SQLSTATEs listed in the 'Diagnostics' section of each function description in Chapter 5, "Functions", on page 67.

Table 173 does not include SQLSTATEs that were remapped between ODBC 2.0 and ODBC 3.0. For a list of SQLSTATEs that were changed in ODBC 3.0, see "SQLSTATE mappings" on page 504.

Note: 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 173. SQLSTATE cross reference

SQLSTATE	Description	Functions
01000	Warning.	<ul style="list-style-type: none"> • SQLAllocHandle() • SQLCloseCursor() • SQLColAttribute() • SQLDescribeParam() • SQLEndTran() • SQLFreeHandle() • SQLGetConnectAttr() • SQLGetStmtAttr() • SQLSetConnectAttr() • SQLSetStmtAttr()
01002	Disconnect error.	<ul style="list-style-type: none"> • SQLDisconnect()
01004	Data truncated.	<ul style="list-style-type: none"> • SQLColAttribute() • SQLDataSources() • SQLDescribeCol() • SQLDriverConnect() • SQLExtendedFetch() • SQLFetch() • SQLGetConnectAttr() • SQLGetCursorName() • SQLGetData() • SQLGetInfo() • SQLGetStmtAttr() • SQLGetSubString() • SQLNativeSql() • SQLPutData() • SQLSetColAttributes()
01504	The UPDATE or DELETE statement does not include a WHERE clause.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
01S00	Invalid connection string attribute.	<ul style="list-style-type: none"> • SQLDriverConnect()
01S01	Error in row.	<ul style="list-style-type: none"> • SQLExtendedFetch()
01S02	Option value changed.	<ul style="list-style-type: none"> • SQLDriverConnect() • SQLSetConnectAttr() • SQLSetStmtAttr()
07001	Wrong number of parameters.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute()
07002	Too many columns.	<ul style="list-style-type: none"> • SQLExtendedFetch() • SQLFetch()
07005	The statement did not return a result set.	<ul style="list-style-type: none"> • SQLColAttribute() • SQLDescribeCol()

SQLSTATE cross reference

Table 173. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
07006	Invalid conversion.	<ul style="list-style-type: none"> • SQLBindParameter() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLGetLength() • SQLGetPosition() • SQLGetSubString() • SQLSetParam()
07009	Invalid descriptor index.	<ul style="list-style-type: none"> • SQLBindCol() • SQLBindParameter() • SQLColAttribute() • SQLDescribeParam() • SQLFetch() • SQLGetData()
08001	Unable to connect to data source.	<ul style="list-style-type: none"> • SQLConnect()
08002	Connection in use.	<ul style="list-style-type: none"> • SQLConnect()
08003	Connection is closed.	<ul style="list-style-type: none"> • SQLAllocHandle() • SQLDisconnect() • SQLEndTran() • SQLFreeHandle() • SQLGetConnectAttr() • SQLGetInfo() • SQLNativeSql() • SQLSetConnectAttr() • SQLSetConnection()
08004	The application server rejected establishment of the connection.	<ul style="list-style-type: none"> • SQLConnect()
08007	Connection failure during transaction.	<ul style="list-style-type: none"> • SQLEndTran()
08S01	Communication link failure.	<ul style="list-style-type: none"> • SQLSetConnectAttr() • SQLSetStmtAttr()
0F001	The LOB token variable does not currently represent any value.	<ul style="list-style-type: none"> • SQLGetLength() • SQLGetPosition() • SQLGetSubString()
21S01	Insert value list does not match column list.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
21S02	Degrees of derived table does not match column list.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
22001	String data right truncation.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPutData()
22002	Invalid output or indicator buffer specified.	<ul style="list-style-type: none"> • SQLExtendedFetch() • SQLFetch() • SQLGetData()
22008	Invalid datetime format or datetime field overflow.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLPutData()
22011	A substring error occurred.	<ul style="list-style-type: none"> • SQLGetSubString()

Table 173. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
22012	Division by zero is invalid.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch()
22018	Error in assignment.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLPutData()
23000	Integrity constraint violation.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute()
24000	Invalid cursor state.	<ul style="list-style-type: none"> • SQLCloseCursor() • SQLColumnPrivileges() • SQLColumns() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLGetData() • SQLGetStmtAttr() • SQLGetTypeInfo() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedureColumns() • SQLProcedures() • SQLSetColAttributes() • SQLSetStmtAttr() • SQLSpecialColumns() • SQLStatistics() • SQLTablePrivileges() • SQLTables()
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute()
25000 25501	Invalid transaction state.	<ul style="list-style-type: none"> • SQLDisconnect()
34000	Invalid cursor name.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare() • SQLSetCursorName()
37XXX	Invalid SQL syntax.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLNativeSql() • SQLPrepare()
40001	Transaction rollback.	<ul style="list-style-type: none"> • SQLEndTran() • SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()

SQLSTATE cross reference

Table 173. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
40003	Communication link failure.	<ul style="list-style-type: none"> • SQLBindCol() • SQLBindParameter() • SQLCancel() • SQLColumnPrivileges() • SQLColumns() • SQLDescribeCol() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLFreeStmt() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetInfo() • SQLGetLength() • SQLGetPosition() • SQLGetSubString() • SQLGetTypeInfo() • SQLMoreResults() • SQLNumParams() • SQLNumResultCols() • SQLParamData() • SQLParamOptions() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedureColumns() • SQLProcedures() • SQLPutData() • SQLRowCount() • SQLSetColAttributes() • SQLSetCursorName() • SQLSetParam() • SQLSpecialColumns() • SQLStatistics() • SQLTablePrivileges() • SQLTables()
42XXX	Syntax error or access rule violation	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
42000	Invalid SQL syntax.	<ul style="list-style-type: none"> • SQLNativeSql()
425XX	Syntax error or access rule violation	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
42601	PARMLIST syntax error.	<ul style="list-style-type: none"> • SQLProcedureColumns()
42818	The operands of an operator or function are not compatible.	<ul style="list-style-type: none"> • SQLGetPosition()
42895	The value of a host variable in the EXECUTE or OPEN statement cannot be used because of its data type	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute()
42S01 ¹	Database object already exists.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
42S02	Database object does not exist.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
42S11	Index already exists.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()

Table 173. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
42S12	Index not found.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
42S21	Column already exists.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
42S22	Column not found.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLPrepare()
44000	Integrity constraint violation.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute()
54028	The maximum number of concurrent LOB handles has been reached.	<ul style="list-style-type: none"> • SQLFetch()
58004	The maximum number of concurrent LOB handles has been reached.	<ul style="list-style-type: none"> • SQLBindCol() • SQLBindParameter() • SQLConnect() • SQLDriverConnect() • SQLDataSources() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLFreeStmt() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetInfo() • SQLGetLength() • SQLGetPosition() • SQLGetSubString() • SQLMoreResults() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetCursorName() • SQLSetParam()
HY000 ²	General error.	<ul style="list-style-type: none"> • SQLAllocHandle() • SQLCloseCursor() • SQLColAttribute() • SQLDescribeParam() • SQLEndTran() • SQLFreeHandle() • SQLGetConnectAttr() • SQLGetStmtAttr() • SQLSetColAttributes() • SQLSetConnection() • SQLSetStmtAttr()
HY001	Memory allocation failure.	All functions.
HY003	Program type out of range.	<ul style="list-style-type: none"> • SQLBindCol() • SQLBindParameter() • SQLGetData() • SQLGetLength() • SQLGetSubString() • SQLSetParam()
HY004	Invalid SQL data type.	<ul style="list-style-type: none"> • SQLBindParameter() • SQLGetTypeInfo() • SQLSetParam()

SQLSTATE cross reference

Table 173. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
HY009	Invalid use of a null pointer.	<ul style="list-style-type: none"> • SQLAllocHandle() • SQLBindParameter() • SQLColumnPrivileges() • SQLExecDirect() • SQLForeignKeys() • SQLGetData() • SQLGetFunctions() • SQLGetInfo() • SQLGetLength() • SQLGetPosition() • SQLNativeSql() • SQLNumParams() • SQLNumResultCols() • SQLPrepare() • SQLPutData() • SQLSetCursorName() • SQLSetConnectAttr() • SQLSetEnvAttr() • SQLSetParam() • SQLSetStmtAttr()
HY010	Function sequence error.	<ul style="list-style-type: none"> • SQLBindCol() • SQLBindParameter() • SQLCloseCursor() • SQLColAttribute() • SQLColumns() • SQLDescribeCol() • SQLDescribeParam() • SQLDisconnect() • SQLEndTran() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLFreeHandle() • SQLFreeStmt() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetStmtAttr() • SQLGetTypeInfo() • SQLMoreResults() • SQLNumParams() • SQLNumResultCols() • SQLParamData() • SQLParamOptions() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedureColumns() • SQLProcedures() • SQLPutData() • SQLRowCount() • SQLSetColAttributes() • SQLSetConnectAttr() • SQLSetCursorName() • SQLSetParam() • SQLSetStmtAttr() • SQLSpecialColumns() • SQLStatistics() • SQLTablePrivileges() • SQLTables()
HY011	Operation invalid at this time.	<ul style="list-style-type: none"> • SQLSetConnectAttr() • SQLSetEnvAttr() • SQLSetStmtAttr()

Table 173. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
HY012	Invalid transaction code.	<ul style="list-style-type: none"> • SQLEndTran()
HY013	Unexpected memory handling error.	<ul style="list-style-type: none"> • SQLAllocHandle() • SQLBindCol() • SQLBindParameter() • SQLCancel() • SQLCloseCursor() • SQLConnect() • SQLDataSources() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLFreeHandle() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetLength() • SQLGetPosition() • SQLGetStmtAttr() • SQLGetSubString() • SQLMoreResults() • SQLNumParams() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetColAttributes() • SQLSetCursorName() • SQLSetParam()
HY014	No more handles.	<ul style="list-style-type: none"> • SQLAllocHandle() • SQLColumnPrivileges() • SQLColumns() • SQLExecDirect() • SQLExecute() • SQLForeignKeys() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedureColumns() • SQLProcedures() • SQLSpecialColumns() • SQLStatistics() • SQLTablePrivileges() • SQLTables()
HY015	No cursor name available.	<ul style="list-style-type: none"> • SQLGetCursorName()
HY019	Numeric value out of range.	<ul style="list-style-type: none"> • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLPutData()
HY024	Invalid argument value.	<ul style="list-style-type: none"> • SQLConnect() • SQLGetSubString() • SQLSetConnectAttr() • SQLSetEnvAttr() • SQLSetStmtAttr()

SQLSTATE cross reference

Table 173. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
HY090	Invalid string or buffer length.	<ul style="list-style-type: none"> • SQLBindCol() • SQLBindParameter() • SQLColAttribute() • SQLColumnPrivileges() • SQLColumns() • SQLConnect() • SQLDataSources() • SQLDescribeCol() • SQLDriverConnect() • SQLExecDirect() • SQLParamData() • SQLForeignKeys() • SQLGetConnectAttr() • SQLGetCursorName() • SQLGetData() • SQLGetInfo() • SQLGetPosition() • SQLGetStmtAttr() • SQLGetSubString() • SQLNativeSql() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedures() • SQLProcedureColumns() • SQLPutData() • SQLSetColAttributes() • SQLSetConnectAttr() • SQLSetCursorName() • SQLSetEnvAttr() • SQLSetStmtAttr() • SQLSpecialColumns() • SQLStatistics() • SQLTables() • SQLTablePrivileges()
HY091	Descriptor type out of range.	<ul style="list-style-type: none"> • SQLColAttribute()
HY092	Option type out of range.	<ul style="list-style-type: none"> • SQLAllocHandle() • SQLEndTran() • SQLFreeStmt() • SQLGetConnectAttr() • SQLGetCursorName() • SQLGetEnvAttr() • SQLGetStmtAttr() • SQLSetConnectAttr() • SQLSetEnvAttr() • SQLSetStmtAttr()
HY096	Information type out of range.	<ul style="list-style-type: none"> • SQLGetInfo()
HY097	Column type out of range.	<ul style="list-style-type: none"> • SQLSpecialColumns()
HY098	Scope type out of range.	<ul style="list-style-type: none"> • SQLSpecialColumns()
HY099	Nullable type out of range.	<ul style="list-style-type: none"> • SQLSpecialColumns()
HY100	Uniqueness option type out of range.	<ul style="list-style-type: none"> • SQLStatistics()
HY101	Accuracy option type out of range.	<ul style="list-style-type: none"> • SQLStatistics()
HY103	Direction option out of range.	<ul style="list-style-type: none"> • SQLDataSources()
HY104	Invalid precision or scale value.	<ul style="list-style-type: none"> • SQLBindParameter() • SQLSetColAttributes() • SQLSetParam()
HY105	Invalid parameter type.	<ul style="list-style-type: none"> • SQLBindParameter()
HY106	Fetch type out of range.	<ul style="list-style-type: none"> • SQLExtendedFetch()
HY107	Row value out of range.	<ul style="list-style-type: none"> • SQLParamOptions()

Table 173. SQLSTATE cross reference (continued)

SQLSTATE	Description	Functions
HY109	Invalid cursor position.	<ul style="list-style-type: none"> SQLGetStmtAttr()
HY110	Invalid driver completion.	<ul style="list-style-type: none"> SQLDriverConnect()
HYC00	Driver not capable.	<ul style="list-style-type: none"> SQLBindCol() SQLBindParameter() SQLColAttribute() SQLColumnPrivileges() SQLColumns() SQLDescribeCol() SQLDescribeParam() SQLExtendedFetch() SQLFetch() SQLForeignKeys() SQLGetConnectAttr() SQLGetData() SQLGetInfo() SQLGetLength() SQLGetPosition() SQLGetStmtAttr() SQLGetSubString() SQLPrimaryKeys() SQLProcedureColumns() SQLProcedures() SQLSetConnectAttr() SQLSetEnvAttr() SQLSetParam() SQLSetStmtAttr() SQLSpecialColumns() SQLStatistics() SQLTables() SQLTablePrivileges()
S1002	Invalid column number.	<ul style="list-style-type: none"> SQLDescribeCol() SQLSetColAttributes()
S1093	Invalid parameter number.	<ul style="list-style-type: none"> SQLSetParam()
S1501	Invalid data source name.	<ul style="list-style-type: none"> SQLConnect()

Note:

- 42Sxx SQLSTATEs replace S00xx SQLSTATEs.
- HYxxx SQLSTATEs replace S1xxx SQLSTATEs.

SQLSTATE cross reference

Appendix D. Data conversion

This section contains tables used for data conversion between C and SQL data types. This includes:

- Precision, scale, length, and display size of each data type
- Conversion from SQL to C data types
- Conversion from C to SQL data types

For a list of SQL and C data types, their symbolic types, and the default conversions, see Table 4 on page 31 and Table 5 on page 32. Supported conversions are shown in Table 8 on page 35.

Identifiers for date, time, and timestamp data types changed in ODBC 3.0. See “Changes to datetime data types” on page 505 for the data type mappings.

Data type attributes

Information is shown for the following data type attributes:

- “Precision”
- “Scale” on page 486
- “Length” on page 487
- “Display size” on page 488

Precision

The precision of a numeric column or parameter refers to the maximum number of digits used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter. The following table defines the precision for each SQL data type.

Data type attributes

Table 174. Precision

fSqlType	Precision
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length of the column or parameter. For example, the precision of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column or parameter. ^a
SQL_DECIMAL SQL_NUMERIC	The defined maximum number of digits. For example, the precision of a column defined as NUMERIC(10,3) is 10.
SQL_SMALLINT ^b	5
SQL_INTEGER ^b	10
SQL_FLOAT ^b	15
SQL_REAL ^b	7
SQL_ROWID	40
SQL_DOUBLE ^b	15
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length of the column or parameter. For example, the precision of a column defined as CHAR(10) FOR BIT DATA, is 10.
SQL_LONGVARBINARY	The maximum length of the column or parameter.
SQL_TYPE_DATE ^b	10 (the number of characters in the yyyy-mm-dd format).
SQL_TYPE_TIME ^b	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.fff[fff]" or "yyyy-mm-dd.hh.mm.ss.fff[fff]" format used by the TIMESTAMP data type. For example, if a timestamp does not use seconds or fractional seconds, the precision is 16 (the number of characters in the "yyyy-mm-dd hh:mm" format). If a timestamp uses thousandths of a second, the precision is 26 (the number of characters in the "yyyy-mm-dd hh:mm:ss.fffff" format). The maximum for fractional seconds is 6 digits.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column or parameter. For example, the precision of a column defined as GRAPHIC(10) is 10.
SQL_LONGVARGRAPHIC	The maximum length of the column or parameter.

Note:

- ^a When defining the precision of a parameter of this data type with SQLBindParameter() or SQLSetParam(), *cbParamDef* should be set to the total length of the data, not the precision as defined in this table.
- ^b The *cbParamDef* argument of SQLBindParameter() or SQLSetParam() is ignored for this data type.

Scale

The scale of a numeric column or parameter refers to the maximum number of digits to the right of the decimal point. Note that, for approximate floating point number columns or parameters, the scale is undefined, since the number of digits to the right of the decimal place is not fixed. The following table defines the scale for each SQL data type.

Table 175. Scale

fSqlType	Scale
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Not applicable.
SQL_DECIMAL SQL_NUMERIC	The defined number of digits to the right of the decimal place. For example, the scale of a column defined as NUMERIC(10, 3) is 3.
SQL_SMALLINT SQL_INTEGER	0
SQL_REAL SQL_FLOAT SQL_DOUBLE	Not applicable.
SQL_ROWID	Not applicable.
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Not applicable.
SQL_TYPE_DATE SQL_TYPE_TIME	Not applicable.
SQL_TYPE_TIMESTAMP	The number of digits to the right of the decimal point in the "yyyy-mm-dd hh:mm:ss[fff[fff]]" format. For example, if the TIMESTAMP data type uses the "yyyy-mm-dd hh:mm:ss.fff" format, the scale is 3. The maximum for fractional seconds is 6 digits.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC SQL_DBCLOB	Not applicable.

Length

The length of a column is the maximum number of *bytes* returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte. Note that the length of a column can be different than the number of bytes required to store the data on the data source. For a list of default C data types, see the "Default C Data Types" section.

The following table defines the length for each SQL data type.

Data type attributes

Table 176. Length

fSqlType	Length
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length of the column. For example, the length of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column.
SQL_DECIMAL SQL_NUMERIC	The maximum number of digits plus two. Since these data types are returned as character strings, characters are needed for the digits, a sign, and a decimal point. For example, the length of a column defined as NUMERIC(10,3) is 12.
SQL_SMALLINT	2 (two bytes).
SQL_INTEGER	4 (four bytes).
SQL_REAL	4 (four bytes).
SQL_ROWID	40
SQL_FLOAT	8 (eight bytes).
SQL_DOUBLE	8 (eight bytes).
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length of the column. For example, the length of a column defined as CHAR(10) FOR BIT DATA is 10.
SQL_LONGVARBINARY	The maximum length of the column.
SQL_TYPE_DATE SQL_TYPE_TIME	6 (the size of the DATE_STRUCT or TIME_STRUCT structure).
SQL_TYPE_TIMESTAMP	16 (the size of the TIMESTAMP_STRUCT structure).
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column times 2. For example, the length of a column defined as GRAPHIC(10) is 20.
SQL_LONGVARGRAPHIC	The maximum length of the column times 2.

Display size

The display size of a column is the maximum number of *bytes* needed to display data in character form. The following table defines the display size for each SQL data type.

Table 177. Display size

fSqlType	Display size
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length of the column. For example, the display size of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column.
SQL_DECIMAL SQL_NUMERIC	The precision of the column plus two (a sign, precision digits, and a decimal point). For example, the display size of a column defined as NUMERIC(10,3) is 12.
SQL_SMALLINT	6 (a sign and 5 digits).
SQL_INTEGER	11 (a sign and 10 digits).
SQL_REAL	13 (a sign, 7 digits, a decimal point, the letter E, a sign, and 2 digits).
SQL_ROWID	40
SQL_FLOAT SQL_DOUBLE	22 (a sign, 15 digits, a decimal point, the letter E, a sign, and 3 digits).
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length of the column times 2 (each binary byte is represented by a 2 digit hexadecimal number). For example, the display size of a column defined as CHAR(10) FOR BIT DATA is 20.
SQL_LONGVARBINARY	The maximum length of the column times 2.
SQL_TYPE_DATE	10 (a date in the format yyyy-mm-dd).
SQL_TYPE_TIME	8 (a time in the format hh:mm:ss).
SQL_TYPE_TIMESTAMP	19 (if the scale of the timestamp is 0) or 20 plus the scale of the timestamp (if the scale is greater than 0). This is the number of characters in the "yyyy-mm-dd hh:mm:ss[fffff]" or "yyyy-mm-dd.hh.mm.ss.fffff" format. For example, the display size of a column storing thousandths of a second is 23 (the number of characters in "yyyy-mm-dd hh:mm:ss.fffff"). The maximum for fractional seconds is 6 digits.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column or parameter. For example, the display size of a column defined as GRAPHIC(10) is 20.
SQL_LONGVARGRAPHIC	The maximum length of the column or parameter.

Converting data from SQL to C data types

For a given SQL data type:

- The first column of the table lists the legal input values of the *fCType* argument in `SQLBindCol()` and `SQLGetData()`.
- The second column lists the outcomes of a test, often using the *cbValueMax* argument specified in `SQLBindCol()` or `SQLGetData()`, which the driver performs to determine if it can convert the data.
- The third and fourth columns list the values (for each outcome) of the *rgbValue* and *pcbValue* arguments specified in the `SQLBindCol()` or `SQLGetData()` after the driver has attempted to convert the data.
- The last column lists the `SQLSTATE` returned for each outcome by `SQLFetch()`, `SQLExtendedFetch()`, or `SQLGetData()`.

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

SQL to C data types

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 S1C00` (driver not capable).

Though it is not shown in the tables, the *pcbValue* argument contains `SQL_NULL_DATA` when the SQL data value is `NULL`. For an explanation of the use of *pcbValue* when multiple calls are made to retrieve data, see `SQLGetData()`.

When SQL data is converted to character C data, the character count returned in *pcbValue* does not include the null termination byte. If *rgbValue* is a null pointer, `SQLBindCol()` or `SQLGetData()` returns `SQLSTATE S1009` (invalid argument value).

In the following tables:

Data length

The total length of the data after it has been converted to the specified C data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is returned to the application.

Significant digits

The minus sign (if needed) and the digits to the left of the decimal point.

Display size

The total number of bytes needed to display data in the character format.

Converting character SQL data to C data

The character SQL data types are:

- SQL_CHAR
- SQL_VARCHAR
- SQL_LONGVARCHAR
- SQL_CLOB

Table 178. Converting character SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Data length < cbValueMax	Data	Data length	00000
	Data length >= cbValueMax	Truncated data	Data length	01004
SQL_C_BINARY	Data length <= cbValueMax	Data	Data length	00000
	Data length > cbValueMax	Truncated data	Data length	01004
SQL_C_SHORT	Data converted without truncation ^a	Data	Size of the C data type	00000
SQL_C_LONG				
SQL_C_FLOAT				
SQL_C_DOUBLE	Data converted with truncation, but without loss of significant digits ^a	Data	Size of the C data type	01004
SQL_C_TINYINT				
SQL_C_BIT	Conversion of data would result in loss of significant digits ^a	Untouched	Size of the C data type	22003
	Data is not a number ^a	Untouched	Size of the C data type	22005
SQL_C_TYPE_DATE	Data value is a valid date ^a	Data	6 ^b	00000
	Data value is not a valid date ^a	Untouched	6 ^b	22008
SQL_C_TYPE_TIME	Data value is a valid time ^a	Data	6 ^b	00000
	Data value is not a valid time ^a	Untouched	6 ^b	22008
SQL_C_TYPE_TIMESTAMP	Data value is a valid timestamp ^a	Data	16 ^b	00000
	Data value is not a valid timestamp ^a	Untouched	16 ^b	22008

Note:

^a The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

^b This is the size of the corresponding C data type.

SQLSTATE 00000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns `SQL_SUCCESS`.

Converting graphic SQL data to C data

The graphic SQL data types are:

SQL_GRAPHIC
 SQL_VARGRAPHIC
 SQL_LONGVARGRAPHIC
 SQL_DBCLOB

SQL to C data types

Table 179. Converting graphic SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Number of double byte characters * 2 <= cbValueMax	Data	Length of data(octets)	00000
	Number of double byte characters * 2 <= cbValueMax	Truncated data, to the nearest even byte that is less than <i>cbValueMax</i> .	Length of data(octets)	01004
SQL_C_DBCHAR	Number of double byte characters * 2 < cbValueMax	Data	Length of data(octets)	00000
	Number of double byte characters * 2 >= cbValueMax	Truncated <i>cbValueMax</i> . data, to the nearest even byte that is less than <i>cbValueMax</i> .	Length of data(octets)	01004

Note:

SQLSTATE 00000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Converting numeric SQL data to C data

The numeric SQL data types are:

SQL_DECIMAL
 SQL_NUMERIC
 SQL_SMALLINT
 SQL_INTEGER
 SQL_REAL
 SQL_FLOAT
 SQL_DOUBLE

Table 180. Converting numeric SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Data length	00000
	Number of significant digits < cbValueMax	Truncated data	Data length	01004
	Number of significant digits >= cbValueMax	Untouched	Data length	22003
SQL_C_SHORT SQL_C_LONG	Data converted without truncation ^a	Data	Size of the C data type	00000
SQL_C_FLOAT SQL_C_DOUBLE SQL_C_TINYINT	Data converted with truncation, but without loss of significant digits ^a	Truncated data	Size of the C data type	01004
SQL_C_BIT	Conversion of data would result in loss of significant digits ^a	Untouched	Size of the C data type	22003

Note:

^a The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

SQLSTATE 00000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Converting binary SQL data to C data

The binary SQL data types are:

SQL_BINARY
SQL_VARBINARY
SQL_LONGVARBINARY
SQL_BLOB

Table 181. Converting binary SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	(Data length) < cbValueMax	Data	Data length	N/A
	(Data length) >= cbValueMax	Truncated data	Data length	01004
SQL_C_BINARY	Data length <= cbValueMax	Data	Data length	N/A
	Data length > cbValueMax	Truncated data	Data length	01004

Converting date SQL data to C data

The date SQL data type is:

SQL_TYPE_DATE

Table 182. Converting date SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 11	Data	10	00000
	cbValueMax < 11	Untouched	10	22003
SQL_C_TYPE_DATE	None ^a	Data	6 ^b	00000
SQL_C_TYPE_TIMESTAMP	None ^a	Data ^c	16 ^b	00000
SQL_C_BINARY	Data length <= cbValueMax	Data	Data length	00000
	Data length > cbValueMax	Untouched	Untouched	22003

Note:

- ^a The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.
- ^b This is the size of the corresponding C data type.
- ^c The time fields of the `TIMESTAMP_STRUCT` structure are set to zero.

SQLSTATE `00000` is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns `SQL_SUCCESS`.

When the date SQL data type is converted to the character C data type, the resulting string is in the "yyyy-mm-dd" format.

Converting time SQL data to C data

The time SQL data type is:

SQL_TYPE_TIME

SQL to C data types

Table 183. Converting time SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 9	Data	8	00000
	cbValueMax < 9	Untouched	8	22003
SQL_C_TYPE_TIME	None ^a	Data	6 ^b	00000
SQL_C_TYPE_TIMESTAMP	None ^a	Data ^c	16 ^b	00000

Note:

- ^a The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.
- ^b This is the size of the corresponding C data type.
- ^c The date fields of the `TIMESTAMP_STRUCT` structure are set to the current system date of the machine that the application is running, and the time fraction is set to zero.

SQLSTATE 00000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns `SQL_SUCCESS`.

When the time SQL data type is converted to the character C data type, the resulting string is in the "hh:mm:ss" format.

Converting timestamp SQL data to C data

The timestamp SQL data type is:
SQL_TYPE_TIMESTAMP

Table 184. Converting timestamp SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Data length	00000
	19 <= cbValueMax <= Display size	Truncated data ^b	Data length	01004
	cbValueMax < 19	Untouched	Data length	22003
SQL_C_TYPE_DATE	None ^a	Truncated data ^c	6 ^e	01004
SQL_C_TYPE_TIME	None ^a	Truncated data ^d	6 ^e	01004
SQL_C_TYPE_TIMESTAMP	None ^a	Data	16 ^e	00000
	Fractional seconds portion of timestamp is truncated. ^a	Data ^b	16	01004

Note:

- ^a The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.
- ^b The fractional seconds of the timestamp are truncated.
- ^c The time portion of the timestamp is deleted.
- ^d The date portion of the timestamp is deleted.
- ^e This is the size of the corresponding C data type.

SQLSTATE 00000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns `SQL_SUCCESS`.

When the timestamp SQL data type is converted to the character C data type, the resulting string is in the "yyyy-mm-dd hh:mm:ss[.fff[fff]]" format (regardless of the precision of the timestamp SQL data type).

Converting row ID SQL data to C data

The row ID SQL data type is:
SQL_ROWID

Table 185. Converting row ID SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Data length < = cbValueMax	Data	Data length	00000
	Data length > cbValueMax	Truncated data	Data length	01004

SQL to C data conversion examples

Table 186. SQL to C data conversion examples

SQL data type	SQL data value	C data type	cbValueMax	rgbValue	SQL STATE
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0 ^a	00000
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0 ^a	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0 ^a	00000
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0 ^a	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	4	---	22003
SQL_DECIMAL	1234.56	SQL_C_FLOAT	Ignored	1234.56	00000
SQL_DECIMAL	1234.56	SQL_C_SHORT	Ignored	1234	01004
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0 ^a	00000
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	10	---	22003
SQL_TYPE_DATE	1992-12-31	SQL_C_TYPE_TIMESTAMP	Ignored	1992,12,31, 0,0,0,0 ^b	00000
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 ^a	00000
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 ^a	01004
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	---	22003

Note:

^a "\0" represents a null termination character.

^b The numbers in this list are the numbers stored in the fields of the `TIMESTAMP_STRUCT` structure.

SQLSTATE `00000` is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns `SQL_SUCCESS`.

Converting data from C to SQL data types

For a given C data type:

- The first column of the table lists the legal input values of the *fSqlType* argument in `SQLBindParameter()` or `SQLSetParam()`.
- The second column lists the outcomes of a test, often using the length of the parameter data as specified in the *pcbValue* argument in `SQLBindParameter()` or `SQLSetParam()`, which the driver performs to determine if it can convert the data.

C to SQL data types

- The third column lists the SQLSTATE returned for each outcome by `SQLExecDirect()` or `SQLExecute()`.

Note: Data is sent to the data source only if the SQLSTATE is 00000 (success).

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

If the *fSqlType* argument in `SQLBindParameter()` or `SQLSetParam()` contains a value not shown in the table for a given C data type, SQLSTATE 07006 is returned (Restricted data type attribute violation).

If the *fSqlType* argument contains a value shown in the table but which specifies a conversion not supported by the driver, `SQLBindParameter()` or `SQLSetParam()` returns SQLSTATE S1C00 (Driver not capable).

If the *rgbValue* and *pcbValue* arguments specified in `SQLBindParameter()` or `SQLSetParam()` are both null pointers, that function returns SQLSTATE S1009 (Invalid argument value).

Data length

The total length of the data after it has been converted to the specified SQL data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is sent to the data source.

Column length

The maximum number of bytes returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte.

Display size

The maximum number of bytes needed to display data in character form.

Significant digits

The minus sign (if needed) and the digits to the left of the decimal point.

Converting character C data to SQL data

The character C data type is:

`SQL_C_CHAR`

Table 187. Converting character C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR	Data length <= Column length	00000
SQL_VARCHAR		
SQL_LONGVARCHAR	Data length > Column length	01004
SQL_CLOB		
SQL_DECIMAL	Data converted without truncation	00000
SQL_NUMERIC		
SQL_SMALLINT	Data converted with truncation, but without loss of significant digits	01004
SQL_INTEGER		
SQL_REAL	Conversion of data would result in loss of significant digits	22003
SQL_FLOAT		
SQL_DOUBLE	Data value is not a numeric value	22005
SQL_BINARY	(Data length) < Column length	N/A
SQL_VARBINARY		
SQL_LONGVARBINARY	(Data length) >= Column length	01004
SQL_BLOB	Data value is not a hexadecimal value	22005
SQL_ROWID	Data length <= Column length	00000
	Data length > Column length	01004
SQL_TYPE_DATE	Data value is a valid date	00000
	Data value is not a valid date	22008
SQL_TYPE_TIME	Data value is a valid time	00000
	Data value is not a valid time	22008
	Data value is a valid timestamp; time portion is non-zero	01004
SQL_TYPE_TIMESTAMP	Data value is a valid timestamp	00000
	Data value is a valid timestamp; fractional seconds portion is non-zero	01004
	Data value is not a valid timestamp	22008
	Data value is a valid timestamp; fractional seconds portion is non-zero	01004
SQL_GRAPHIC	Data length / 2 <= Column length	00000
SQL_VARGRAPHIC		
SQL_LONGVARGRAPHIC	Data length / 2 < Column length	01004
SQL_DBCLOB		

Note:

SQLSTATE 00000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Converting numeric C data to SQL data

The numeric C data types are:

```
SQL_C_SHORT
SQL_C_LONG
SQL_C_FLOAT
SQL_C_DOUBLE
SQL_C_TINYINT
SQL_C_BIT
```

C to SQL data types

Table 188. Converting numeric C data to SQL data

fSQLType	Test	SQLSTATE
SQL_DECIMAL SQL_NUMERIC	Data converted without truncation	00000
SQL_SMALLINT SQL_INTEGER SQL_REAL	Data converted with truncation, but without loss of significant digits	01004
SQL_FLOAT SQL_DOUBLE	Conversion of data would result in loss of significant digits	22003
SQL_CHAR SQL_VARCHAR	Data converted without truncation.	00000
	Conversion of data would result in loss of significant digits.	22003

Note:

SQLSTATE 00000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Converting binary C data to SQL data

The binary C data type is:
SQL_C_BINARY

Table 189. Converting binary C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Data length <= Column length	N/A
SQL_LONGVARCHAR SQL_CLOB	Data length > Column length	01004
SQL_BINARY SQL_VARBINARY	Data length <= Column length	N/A
SQL_LONGVARBINARY SQL_BLOB	Data length > Column length	01004

Converting DBCHAR C data to SQL data

The double byte C data type is:
SQL_C_DBCHAR

Table 190. Converting DBCHAR C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Data length <= Column length x 2	N/A
SQL_LONGVARCHAR SQL_CLOB	Data length > Column length x 2	01004
SQL_BINARY SQL_VARBINARY	Data length <= Column length x 2	N/A
SQL_LONGVARBINARY SQL_BLOB	Data length > Column length x 2	01004

Converting date C data to SQL data

The date C data type is:
SQL_C_TYPE_DATE

Table 191. Converting date C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR	Column length >= 10	00000
SQL_VARCHAR	Column length < 10	22003
SQL_TYPE_DATE	Data value is a valid date	00000
	Data value is not a valid date	22008
SQL_TYPE_TIMESTAMP ^a	Data value is a valid date	00000
	Data value is not a valid date	22008

Note:

^a The time component of TIMESTAMP is set to zero.

SQLSTATE 00000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Converting time C data to SQL data

The time C data type is:
SQL_C_TYPE_TIME

Table 192. Converting time C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR	Column length >= 8	00000
SQL_VARCHAR	Column length < 8	22003
SQL_TYPE_TIME	Data value is a valid time	00000
	Data value is not a valid time	22008
SQL_TYPE_TIMESTAMP ^a	Data value is a valid time	00000
	Data value is not a valid time	22008

Note:

^a The date component of TIMESTAMP is set to the system date of the machine at which the application is running.

SQLSTATE 00000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Converting timestamp C data to SQL data

The timestamp C data type is:
SQL_TYPE_C_TIMESTAMP

C to SQL data types

Table 193. Converting timestamp C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR	Column length >= Display size	00000
SQL_VARCHAR	19 <= Column length < Display size ^a	01004
	Column length < 19	22003
SQL_TYPE_DATE	Data value is a valid date ^b	01004
	Data value is not a valid date	22008
SQL_TYPE_TIME	Data value is a valid time ^c	01004
	Data value is not a valid time	22008
	Fractional seconds fields are non-zero	01004
SQL_TYPE_TIMESTAMP	Data value is a valid timestamp	00000
	Data value is not a valid timestamp	22008

Note:

^a The fractional seconds of the timestamp are truncated.

^b The time portion of the timestamp is deleted.

^c The date portion of the timestamp is deleted.

SQLSTATE 00000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

C to SQL data conversion examples

Table 194. 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	00000
SQL_C_CHAR	abcde\0	SQL_CHAR	5	abcde	01004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6	1234.56	00000
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	5	1234.5	01004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	3	---	22003
SQL_C_FLOAT	1234.56	SQL_FLOAT	Not applicable	1234.56	00000
SQL_C_FLOAT	1234.56	SQL_INTEGER	Not applicable	1234	01004

Note:

SQLSTATE 00000 is not returned by SQLGetDiagRec(), rather it is indicated when the function returns SQL_SUCCESS.

Appendix E. Deprecated function

This section explains DB2 for OS/390 and z/OS's support of the ODBC 3.0 standard. DB2 ODBC introduces ODBC 3.0 support in Version 6.

Mapping deprecated functions

The ODBC 3.0 functions replace, or deprecate, many existing ODBC 2.0 functions. The DB2 ODBC driver continues to support all of the deprecated functions.

Recommendation: Begin using ODBC 3.0 functional replacements to maintain optimum portability.

Table 195 lists the ODBC 2.0 deprecated functions and the ODBC 3.0 replacement functions.

Table 195. ODBC 2.0 deprecated functions

ODBC 2.0 deprecated function	Purpose	ODBC 3.0 replacement function
SQLAllocEnv()	Obtains an environment handle.	SQLAllocHandle() with <i>handletype</i> =SQL_HANDLE_ENV
SQLAllocConnect()	Obtains a connection handle.	SQLAllocHandle() with <i>handletype</i> =SQL_HANDLE_DBC
SQLAllocStmt()	Obtains a statement handle.	SQLAllocHandle() with <i>handletype</i> =SQL_HANDLE_STMT
SQLColAttributes()	Gets column attributes.	SQLColAttribute()
SQLError()	Returns additional diagnostic information (multiple fields of the diagnostic data structure).	SQLGetDiagRec()
SQLFreeEnv()	Frees environment handle.	SQLFreeHandle() with <i>handletype</i> =SQL_HANDLE_ENV
SQLFreeConnect()	Frees connection handle.	SQLFreeHandle() with <i>handletype</i> =SQL_HANDLE_DBC
SQLGetConnectOption()	Returns a value of a connection attribute.	SQLGetConnectAttr()
SQLGetStmtOption()	Returns a value of a statement attribute.	SQLGetStmtAttr()
SQLSetConnectOption()	Sets a value of a connection attribute.	SQLSetConnectAttr()
SQLSetStmtOption()	Sets a value of a statement attribute.	SQLSetStmtAttr()
SQLTransact()	Commits or rolls back a transaction.	SQLEndTran()

Note: The SQL_DROP option is deprecated for SQLFreeStmt(). SQLFreeHandle() with *handletype* = SQL_HANDLE_STMT replaces the SQL_DROP option.

Changes to SQLGetInfo information types

Values of the *InfoType* arguments for SQLGetInfo() arguments were renamed in ODBC 3.0. Table 88 on page 255 lists the ODBC 2.0 and ODBC 3.0 argument names.

Changes to SQLSetConnectAttr attributes

The renamed ODBC 3.0 attribute values support all existing ODBC 2.0 attributes. You can specify either the ODBC 2.0 or the ODBC 3.0 attribute value; the ODBC driver supports both. Table 196 matches ODBC 2.0 and ODBC 3.0 values.

Table 196. SQLSetConnectAttr attribute value mapping

ODBC 2.0 attribute	ODBC 3.0 attribute
SQL_ACCESS_MODE	SQL_ATTR_ACCESS_MODE
SQL_AUTOCOMMIT	SQL_ATTR_AUTOCOMMIT
SQL_CONNECTTYPE	SQL_ATTR_CONNECTTYPE
SQL_CURRENT_SCHEMA	SQL_ATTR_CURRENT_SCHEMA
SQL_MAXCONN	SQL_ATTR_MAXCONN
SQL_PARAMOPT_ATOMIC	SQL_ATTR_PARAMOPT_ATOMIC
SQL_SYNC_POINT	SQL_ATTR_SYNC_POINT
SQL_TXN_ISOLATION	SQL_ATTR_TXN_ISOLATION

Changes to SQLSetEnvAttr attributes

Table 197 lists the SQLSetEnvAttr() attribute values renamed in ODBC 3.0. The ODBC 3.0 attributes support all of the existing ODBC 2.0 attributes. You can specify either the ODBC 2.0 or the ODBC 3.0 attribute value; the ODBC driver supports both.

Table 197. SQLSetEnvAttr attribute value mapping

ODBC 2.0 attribute	ODBC 3.0 attribute
SQL_CONNECTTYPE	SQL_ATTR_CONNECTTYPE
SQL_MAXCONN	SQL_ATTR_MAXCONN
SQL_OUTPUT_NTS	SQL_ATTR_OUTPUT_NTS

Changes to SQLSetStmtAttr attributes

Table 198 lists the SQLSetStmtAttr() attribute values renamed in ODBC 3.0. The ODBC 3.0 attributes support all of the existing ODBC 2.0 attributes. You can specify either the ODBC 2.0 or the ODBC 3.0 attribute value; the ODBC driver supports both.

Table 198. SQLSetStmtAttr attribute value mapping

ODBC 2.0 attribute	ODBC 3.0 attribute
SQL_BIND_TYPE	SQL_ATTR_BIND_TYPE / 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

Table 198. SQLSetStmtAttr attribute value mapping (continued)

ODBC 2.0 attribute	ODBC 3.0 attribute
SQL_NOSCAN	SQL_ATTR_NOSCAN
SQL_RETRIEVE_DATA	SQL_ATTR_RETRIEVE_DATA
SQL_ROWSET_SIZE	SQL_ATTR_ROWSET_SIZE / SQL_ATTR_ROW_ARRAY_SIZE
SQL_STMTTXN_ISOLATION or SQL_TXN_ISOLATION	SQL_ATTR_STMTTXN_ISOLATION or SQL_ATTR_TXN_ISOLATION

ODBC 3.0 driver behavior

Behavioral changes refer to functionality that varies depending on the version of ODBC in use. The ODBC2.0 and ODBC 3.0 drivers behave according to the setting of the SQL_ATTR_ODBC_VERSION environment attribute.

The SQL_ATTR_ODBC_VERSION environment attribute controls whether the DB2 ODBC 3.0 driver exhibits ODBC 2.0 or ODBC 3.0 behavior. This value is implicitly set by the ODBC driver by application calls to the ODBC 3.0 function SQLAllocHandle() or the ODBC 2.0 function SQLAllocEnv(). The application can explicitly set by calls to SQLSetEnvAttr().

- ODBC 3.0 applications first call SQLAllocHandle() to get the environmental handle. The DB2 ODBC 3.0 driver *implicitly* sets SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC3. This setting ensures that ODBC 3.0 applications get ODBC 3.0 behavior.

An ODBC 3.0 application should not invoke SQLAllocHandle() and then call SQLAllocEnv(). Doing so implicitly resets the application to ODBC 2.0 behavior. To avoid resetting an application to ODBC 2.0 behavior, ODBC 3.0 applications should always use SQLAllocHandle() to manage environment handles.

- ODBC 2.0 applications first call SQLAllocEnv() to get the environmental handle. The DB2 ODBC 2.0 driver *implicitly* sets SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC2. This setting ensures that ODBC 2.0 applications get ODBC 2.0 behavior.

An application can verify the ODBC version setting by calling SQLGetEnvAttr() for attribute SQL_ATTR_ODBC_VERSION. An application can explicitly set the ODBC version setting by calling SQLSetEnvAttr() for attribute SQL_ATTR_ODBC_VERSION.

Forward compatibility does not affect ODBC 2.0 applications that were compiled using the previous DB2 ODBC 2.0 driver header files, or ODBC 2.0 applications that are recompiled using the new ODBC 3.0 header files. These applications can continue executing as ODBC 2.0 applications on the DB2 ODBC 3.0 driver. These ODBC 2.0 application need not call SQLSetEnvAttr(). As stated above, when the existing ODBC 2.0 application calls SQLAllocEnv() (ODBC 2.0 API to allocate environment handle), the DB2 ODBC 3.0 driver will implicitly set SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC2. This will ensure ODBC 2.0 driver behavior when using the DB2 ODBC 3.0 driver

SQLSTATE mappings

With an ODBC 3.0 driver when `SQLGetDiagRec()` or `SQLError()` is called several SQLSTATEs will differ:

- HYxxx SQLSTATEs replace S1xxx SQLSTATEs
- 42Sxx SQLSTATEs replace S00xx 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.

Table 199 lists ODBC 2.0 to ODBC 3.0 SQLSTATE mappings.

Table 199. ODBC 2.0 to ODBC 3.0 SQLSTATE mappings

ODBC 2.0 SQLSTATE	ODBC 3.0 SQLSTATE
22003	HY019
22007	22008
22005	22018
37000	42000
S0001	42S01
S0002	42S02
S0011	42S11
S0012	42S12
S0021	42S21
S0022	42S22
S0023	42S23
S1000	HY000
S1001	HY001
S1002	07009
	ODBC 2.0 SQLSTATE S1002 is mapped to ODBC 3.0 SQLSTATE 07009 if the underlying function is one of the following:
	<ul style="list-style-type: none"> • <code>SQLBindCol()</code> • <code>SQLColAttribute()</code> • <code>SQLExtendedFetch()</code> • <code>SQLFetch()</code> • <code>SQLGetData()</code>
S1003	HY003
S1004	HY004
S1009	HY009 HY024
S1010	HY007 HY010
	Mapped to HY007 when <code>SQLDescribeCol()</code> is called prior to calling <code>SQLPrepare()</code> , <code>SQLExecDirect()</code> , or a catalog function for the <i>StatementHandle</i> . Otherwise S1010 is mapped to HY010.
S1011	HY011
S1012	HY012
S1013	HY013

Table 199. ODBC 2.0 to ODBC 3.0 SQLSTATE mappings (continued)

ODBC 2.0 SQLSTATE	ODBC 3.0 SQLSTATE
S1014	HY014
S1015	HY015
S1019	HY019
S1090	HY090
S1091	HY091
S1092	HY092
S1093	07009
	07009 is mapped to S1093 if the underlying function is SQLBindParameter() or SQLDescribeParam().
S1096	HY096
S1097	HY097
S1098	HY098
S1099	HY099
S1100	HY100
S1101	HY101
S1103	HY103
S1104	HY104
S1105	HY105
S1106	HY106
S1107	HY107
S1110	HY110
S1C00	HYC00

Changes to datetime data types

In ODBC 3.0, the identifiers for date, time, and timestamp identifiers have changed. The #defines in the include file `sqlcli1.h` are added for the values defined in Table 201 on page 506:

- For input, either ODBC 2.0 or ODBC 3.0 datetime values can be used with the DB2 ODBC 3.0 driver.
- On output, the DB2 ODBC 3.0 driver determines the appropriate value to return based on the setting of the `SQL_ATTR_ODBC_VERSION` environment attribute.
 - If `SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC2`, the output datetime values are the ODBC 2.0 values.
 - If `SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC3`, the output datetime values are the ODBC 3.0 values.

Table 200. Datetime data type mappings-SQL type identifiers

ODBC 2.0	ODBC 3.0
SQL_DATE(9)	SQL_TYPE_DATE(91)
SQL_TIME(10)	SQL_TYPE_TIME(92)
SQL_TIMESTAMP(11)	SQL_TYPE_TIMETAMP(93)

Table 201. Datetime data type mappings-C type identifiers

ODBC 2.0	ODBC 3.0
SQL_C_DATE(9)	SQL_C_TYPE_DATE(91)
SQL_C_TIME(10)	SQL_C_TYPE_TIME(92)
SQL_C_TIMESTAMP(11)	SQL_C_TYPE_TIMESTAMP(93)

The datetime data type changes affect the following functions:

- SQLBindCol()
- SQLBindParameter()
- SQLColAttribute()
- SQLColumns()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLGetData()
- SQLGetTypeInfo()
- SQLProcedureColumns()
- SQLStatistics()
- SQLSpecialColumns()

Appendix F. Example code

| This section provides DB2 ODBC samples:

- | • “DSN803VP sample application” on page 508 shows the sample verification
| program DSN803VP. You can use this sample to verify that your DB2 ODBC 3.0
| installation is correct. See “Application preparation and execution steps” on
| page 50 for more information about sample applications.
- | • In “Client application calling a DB2 ODBC stored procedure” on page 514, a
| client application (APD29) calls a DB2 ODBC stored procedure (SPD29). It
| includes very fundamental processing of query result sets (a query cursor
| opened in a stored procedure and return to client for fetching). For completeness,
| the CREATE TABLE, data INSERTs and CREATE PROCEDURE definition is
| provided.

DSN8O3VP sample application

```

/*****
/* DB2 ODBC 3.0 installation cerification 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("DSN803VP INITIALIZATION\n");

printf("DSN803VP SQLAllocHandle-Environment\n");
henv=0;
rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv );
CHECK_HANDLE( SQL_HANDLE_ENV, henv, rc );
printf("DSN803VP-henv=%i\n",henv);
printf("DSN803VP SQLAllocHandle-Environment successful\n");

printf("DSN803VP SQLAllocHandle-Connection\n");
hdbc=0;
rc=SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc);
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );
printf("DSN803VP-hdbc=%i\n",hdbc);
printf("DSN803VP SQLAllocHandle-Connection successful\n");

printf("DSN803VP 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("DSN803VP successfully issued a SQLconnect\n");

printf("DSN803VP SQLAllocHandle-Statement\n");
hstmt=0;
rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
printf("DSN803VP hstmt=%i\n",hstmt);
printf("DSN803VP SQLAllocHandle-Statement successful\n");

printf("DSN803VP SQLExecDirect\n");
strcpy((char *)sqlstmt,"SELECT * FROM SYSIBM.SYSDUMMY1");
printf("DSN803VP sqlstmt=%s\n",sqlstmt);
rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
printf("DSN803VP successfully issued a SQLExecDirect\n");

/* sample fetch without looking at values */
printf("DSN803VP SQLFetch\n");
rc=SQLFetch(hstmt);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
printf("DSN803VP successfully issued a SQLFetch\n");

printf("DSN803VP SQLEndTran-Commit\n");
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
printf("DSN803VP SQLEndTran-Commit successful\n");

printf("DSN803VP SQLFreeHandle-Statement\n");
rc=SQLFreeHandle(SQL_HANDLE_STMT,hstmt);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
hstmt=0;
printf("DSN803VP SQLFreeHandle-Statement successful\n");
}

```



```

/*****
/* 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("DSN803VP entry check_error rtn\n");

    switch (frc) {
    case SQL_SUCCESS:
        break;
    case SQL_INVALID_HANDLE:
        printf("DSN803VP check_error> SQL_INVALID_HANDLE\n");
        break;
    case SQL_ERROR:
        printf("DSN803VP check_error> SQL_ERROR\n");
        break;
    case SQL_SUCCESS_WITH_INFO:
        printf("DSN803VP check_error> SQL_SUCCESS_WITH_INFO\n");
        break;
    case SQL_NO_DATA_FOUND:
        printf("DSN803VP check_error> SQL_NO_DATA_FOUND\n");
        break;
    default:
        printf("DSN803VP check_error> Received rc from api rc=%i\n",frc);
        break;
    } /*end switch*/

    print_error(htype,hndl,frc,line,file);

    printf("DSN803VP SQLGetSQLCA\n");
    rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    if( rc == SQL_SUCCESS )
        prt_sqlca();
    else
        printf("DSN803VP check_error SQLGetSQLCA failed rc=%i\n",rc);

    printf("DSN803VP exit check_error rtn\n");
    return (frc);

} /* end check_error */

```

Example code

```

/*****
/* 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("DSN803VP entry print_error rtn\n");

    printf("DSN803VP 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( "DSN803VP SQLSTATE: %s\n", sqlstate ) ;
        printf( "DSN803VP Native Error Code: %ld\n", sqlcode ) ;
        printf( "DSN803VP buffer: %s \n", buffer ) ;
        i++ ;
    }
    printf( ">-----\n" ) ;
    printf("DSN803VP exit print_error rtn\n");
    return( SQL_ERROR ) ;
} /* end print_error */

```

```

/*****
/* prt_sqlca
/*****
SQLRETURN
prt_sqlca()
{
    int i;
    printf("DSN803VP entry prt_sqlca rtn\n");

    printf("\r\nDSN803VP*** Printing the SQLCA:\r");
    printf("\nDSN803VP SQLCAID ... %s",sqlca.sqlcaid);
    printf("\nDSN803VP SQLCABC ... %d",sqlca.sqlcabc);
    printf("\nDSN803VP SQLCODE ... %d",sqlca.sqlcode);
    printf("\nDSN803VP SQLERRML ... %d",sqlca.sqlerrml);
    printf("\nDSN803VP SQLERRMC ... %s",sqlca.sqlerrmc);
    printf("\nDSN803VP SQLERRP ... %s",sqlca.sqlerrp);
    for (i = 0; i < 6; i++)
        printf("\nDSN803VP SQLERRD%d ... %d",i+1,sqlca.sqlerrd??(i??));
    for (i = 0; i < 10; i++)
        printf("\nDSN803VP SQLWARN%d ... %c",i,sqlca.sqlwarn[i]);
    printf("\nDSN803VP SQLWARNA ... %c",sqlca.sqlwarn[10]);
    printf("\nDSN803VP SQLSTATE ... %s",sqlca.sqlstate);

    printf("\nDSN803VP exit prt_sqlca rtn\n");
    return(0);
} /* End of prt_sqlca */

/*****
/* END DSN803VP
/*****

```

Client application calling a DB2 ODBC stored procedure

STEP 1. Create table

```

printf("\nAPDDL SQLExecDirect stmt=%d",__LINE__);
strcpy((char *)sqlstmt,
"CREATE TABLE TABLE2A (INT4 INTEGER,SMINT SMALLINT,FLOAT8 FLOAT");
strcat((char *)sqlstmt,
",DEC312 DECIMAL(31,2),CHR10 CHARACTER(10),VCHR20 VARCHAR(20)");
strcat((char *)sqlstmt,
",LVCHR LONG VARCHAR,CHRSB CHAR(10),CHRBIT CHAR(10) FOR BIT DATA");
strcat((char *)sqlstmt,
",DDATE DATE,TTIME TIME,TSTMP TIMESTAMP)");
printf("\nAPDDL sqlstmt=%s",sqlstmt);
rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

```

STEP 2. Insert 101 rows into table

```

/* insert 100 rows into table2a */
for (jx=1;jx<=100 ;jx++ ) {
printf("\nAPDIN SQLExecDirect stmt=%d",__LINE__);
strcpy((char *)sqlstmt,"insert into table2a values(");
sprintf((char *)sqlstmt+strlen((char *)sqlstmt),"%ld",jx);
strcat((char *)sqlstmt,
",4,8.2E+30,1515151515151.51,'CHAR','VCHAR','LVCCHAR','SBCS'");
strcat((char *)sqlstmt,
",'MIXED','01/01/1991','3:33 PM','1999-09-09-09.09.09.090909')");
printf("\nAPDIN sqlstmt=%s",sqlstmt);
rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
} /* endfor */

```

STEP 3. Define stored procedure with CREATE PROCEDURE SQL statement

```

CREATE PROCEDURE SPD29
(INOUT INTEGER)
PROGRAM TYPE MAIN
EXTERNAL NAME SPD29
COLLID DSNAOCLI
LANGUAGE C
RESULT SET 2
MODIFIES SQL DATA
PARAMETER STYLE GENERAL
NO WLM ENVIRONMENT;

```

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 pcpars=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];

```

Example code

```
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(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
    /*****
SQLINTEGER
main(SQLINTEGER argc, SQLCHAR *argv[] )
{
    printf("\nSPD29 INITIALIZATION");
    scale = 0;
    rc=0;

    rc=0;
    SPCODE=0;

    /* argv0 = sp module name */
    if (argc != 2)
    {
        printf("SPD29 parm number error\n  ");
        printf("SPD29 EXPECTED =%d\n",3);
        printf("SPD29 received =%d\n",argc);
        goto dberror;
    }
    strcpy((char *)spname,(char *)argv[0]);
    result = strcmp((char *)spname,"SPD29",5);
    if (result != 0)
    {
        printf("SPD29 argv0 sp name error\n  ");
        printf("SPD29 compare rusult =%i\n",result);
        printf("SPD29 expected =%s\n","SPD29");
        printf("SPD29 received spname=%s\n",spname);
        printf("SPD29 received argv0 =%s\n",argv[0]);
        goto dberror;
    }
    /* get input scode value */
    SPCODE    = *(SQLINTEGER *) argv[1];
    printf("\nSPD29 SQLAllocHandle-Environment number=    1\n");
    henv=0;
    rc =SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    CHECK_HANDLE (SQL_HANDLE_ENV, henv, rc);
    printf("\nSPD29-henv=%i",henv);
    /*****
    printf("\nSPD29 SQLAllocHandle-Connection ");
    hdbc=0;
    rc=SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc);
    CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);
    printf("\nSPD29-hdbc=%i",hdbc);

```

Example code

```

/*****
/* Make sure no autocommits after cursors are allocated, commits */
/* cause sp failure. No-autocommit could also be specified in the*/
/* INI file.
/* Also, sp could be defined with COMMIT_ON_RETURN in the
/* DB2 catalog table SYSIBM.SYSPROCEDURES, but be wary that this */
/* removes control from the client appl to control commit scope. */
/*****
printf("\nSPD29 SQLSetConnectAttr-no autocommits in stored procs");
rc =SQLSetConnectAttr(hdbc, SQL_ATR_AUTOCOMMIT,
    (void*)SQL_AUTOCOMMIT_OFF, SQL_NTS);
CHECK_HANDLE(SQL_HANDLE_DBC, hdbc, rc);
/*****
printf("\nSPD29 SQLConnect NULL connect in stored proc ");
strcpy((char *)uid,"cliuser");
strcpy((char *)pwd,"password");
printf("\nSPD29 server=%s",NULL);
printf("\nSPD29 uid=%s",uid);
printf("\nSPD29 pwd=%s",pwd);
rc=SQLConnect(hdbc, NULL, 0, uid, SQL_NTS, pwd, SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);
/*****
/* 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 SQLAllocHandle-Statement \n");
    hstmt=0;
    rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
    CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
    printf("\nSPD29-hstmt=%i\n",hstmt);

    printf("\nSPD29 SQLPrepare \n");
    strcpy((char *)sqlstmt,
        "insert into TABLE2A(int4) values(?)");
    printf("\nSPD29 sqlstmt=%s",sqlstmt);
    rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
    CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

    printf("\nSPD29 SQLNumParams \n");
    rc=SQLNumParams(hstmt,&pcpar);
    CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
    if (pcpar!=1) {
        printf("\nSPD29 incorrect pcpar=%d",pcpar);
        goto dberror;
    }
}

```

Example code

```

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);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nSPD29 SQLExecute                \n");
rc=SQLExecute(hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nSPD29 SQLFreeHandle-Statement \n");
rc=SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
/*****
printf("\nAPDIN SQLAllocHandle-Statement  stmt=%d",__LINE__);
hstmt=0;
rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
printf("\nAPDIN-hstmt=%i\n",hstmt);

jx=201;
printf("\nAPDIN SQLExecDirect  stmt=%d",__LINE__);
strcpy((char *)sqlstmt,"insert into table2a values(");
sprintf((char *)sqlstmt+strlen((char *)sqlstmt),"%ld",jx);
strcat((char *)sqlstmt,
",4,8.2E+30,1515151515151.51,'CHAR','VCHAR','LVCCHAR','SBCS'");
strcat((char *)sqlstmt,
", 'MIXED','01/01/1991','3:33 PM','1999-09-09-09.09.09.090909')");
printf("\nAPDIN sqlstmt=%s",sqlstmt);
rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

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 queryresultset */
printf("\nSPD29 SQLAllocHandle-Statement \n");
hstmt=0;
rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
printf("\nSPD29-hstmt=%i\n",hstmt);

printf("\nSPD29 SQLPrepare                \n");
strcpy((char *)sqlstmt,
"SELECT INT4 FROM TABLE2A");
printf("\nSPD29 sqlstmt=%s",sqlstmt);
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nSPD29 SQLExecute                \n");
rc=SQLExecute(hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

```

Example code

```
/* allocate 2nd stmt handle for 2nd queryresultset */
/* generate 2nd queryresultset */
printf("\nSPD29 SQLAllocHandle-Statement \n");
hstmt=0;
rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt2);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt2, rc);
printf("\nSPD29-hstmt2=%i\n",hstmt2);

printf("\nSPD29 SQLPrepare                                     \n");
strcpy((char *)sqlstmt2,
"SELECT CHR10 FROM TABLE2A");
printf("\nSPD29 sqlstmt2=%s",sqlstmt2);
rc=SQLPrepare(hstmt2,sqlstmt2,SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt2, rc);

printf("\nSPD29 SQLExecute                                     \n");
rc=SQLExecute(hstmt2);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt2, rc);

/*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...RECIEVED SPCODE=%i\n",SPCODE);
goto dberror;
break;
}
*/
/*****
/* End SQL statements *****/
/*****
/*Be sure NOT to put a SQLTransact with SQL_COMMIT in a DB2/MVS */
/* stored procedure. Commit is not allowed in a DB2/MVS */
/* 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);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);
/*****
printf("\nSPD29 SQLFreeHandle-Connection number=5 \n");
rc =SQLFreeHandle (SQL_HANDLE_DBC, hdbc);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);
/*****
printf("\nSPD29 SQLFreeHandle-Environment number=6 \n");
rc =SQLFreeHandle (SQL_HANDLE_ENV, henv);
CHECK_HANDLE (SQL_HANDLE_ENV, henv, rc);
/*****
goto pgmend;

dberror:
printf("\nSPD29 entry dberror label");
printf("\nSPD29 rc=%d",rc);
printf("\nSPD29 SQLFreeHandle-Environment number=7 \n");
rc =SQLFreeHandle(SQL_HANDLE_ENV, henv, rc);
```

```

printf("\nSPD29 rc=%d",rc);
rc=12;
rc=12;
SPCODE=12;
goto pgmend;

pgmend:

printf("\nSPD29  TERMINATION  ");
if (rc!=0)
{
printf("\nSPD29 WAS NOT SUCCESSFUL");
printf("\nSPD29 SPCODE = %i", SPCODE );
printf("\nSPD29 rc = %i", rc );
}
else
{
printf("\nSPD29 WAS SUCCESSFUL");
}
/* assign output srcode value */
*(SQLINTEGER *) argv[1] = SPCODE;
exit;
} /*END MAIN*/
/*****
** check_error - call print_error(), checks severity of return code
*****/
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("\nSPD29 entry check_error rtn");

    switch (frc){
    case SQL_SUCCESS:
        break;
    case SQL_INVALID_HANDLE:
        printf("\nSPD29 check_error>SQL_INVALID HANDLE ");
        break;
    case SQL_ERROR:
        printf("\nSPD29 check_error>SQL_ERROR ");
        break;
    case SQL_SUCCESS_WITH_INFO:
        printf("\nSPD29 check_error>SQL_SUCCESS_WITH_INFO");
        break;
    case SQL_NO_DATA_FOUND:
        printf("\nSPD29 check_error>SQL_NO_DATA_FOUND ");
        break;
    default:
        printf("\nSPD29 check_error>Invalid rc from api rc=%i",frc);
        break;
    }/*end switch*/

```

Example code

```
print_error(htype,hndl,frc,line,file);
printf("\nSPD29 SQLGetSQLCA ");
rc =SQLGetSQLCA(henv,hdbc,hstmt,&sqlca);
if(rc ==SQL_SUCCESS )
    prt_sqlca();
else
    printf("\n SPD29-check_error SQLGetSQLCA failed rc=%i",rc);
return (frc);
}
/*****
/* print_error
/* calls SQLGetDiagRec() and 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("\nSPD29 entry print_error rtn");
    printf("\nSPD29 rc=%d reported fro file:%s,line:%d --- ",
           frc,
           file,
           line
           );
    i = 1;
    while (SQLGetDiagRec( htype,
                        hndl,
                        i,
                        sqlstate,
                        &sqlcode,
                        buffer,
                        SQL_MAX_MESSAGE_LENGTH + 1,
                        &length
                        ) == SQL_SUCCESS ){
        printf("\nSPD29 SQLSTATE: %s", sqlstate);
        printf("\nSPD29 Native Error Code: %ld", sqlcode);
        printf("\nSPD29 buffer: %s", buffer);
        i++;
    }
    printf("\n>-----");
    printf("\nSPD29 exit print_error rtn");
    return(SQL_ERROR);
}
/*****
/*          P r i n t   S Q L C A
*****/
SQLRETURN
prt_sqlca()
{
    SQLINTEGER i;
    printf("\n\SPD29 entry prts_sqlca rtn");
    printf("\r\r*** Printing the SQLCA:\r");
    printf("\nSQLCAID .... %s",sqlca.sqlcaid);
    printf("\nSQLCABC .... %d",sqlca.sqlcabc);
    printf("\nSQLCODE .... %d",sqlca.sqlcode);
    printf("\nSQLERRML ... %d",sqlca.sqlerrml);
    printf("\nSQLERRMC ... %s",sqlca.sqlerrmc);
    printf("\nSQLERRP ... %s",sqlca.sqlerrp);
    for (i = 0; i < 6; i++)
        printf("\nSQLERRD%d ... %d",i+1,sqlca.sqlerrd??(i??));
    for (i = 0; i < 10; i++)
        printf("\nSQLWARN%d ... %c",i,sqlca.sqlwarn[i]);
    printf("\nSQLWARNA ... %c",sqlca.sqlwarn[10]);
    printf("\nSQLSTATE ... %s",sqlca.sqlstate);
}
```

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 VERIFY) */
/* -SPCODE=1 */
/* -PRINTF MSGS (CHECK SDSF FOR SPAS ADDR TO VERIFY) */
/* -SQLPREPARE/EXECUTE INSERT INT4=200 */
/* -SQLEXECDIRECT INSERT INT4=201 */
/* -SPCODE=2 */
/* -PRINTF MSGS (CHECK SDSF FOR SPAS ADDR TO VERIFY) */
/* -SQLPREPARE/EXECUTE SELECT INT4 FROM TABLE2A */
/* -SQLPREPARE/EXECUTE SELECT CHR10 FROM TABLE2A */
/* (CLI CURSORS OPENED 'WITH RETURN')... */
/* -RETURN */
/* -FETCH QRS FROM SP CURSOR */
/* -COMMIT */
/* -VERIFY 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 */
/*****
#ifdef 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 pcpa=0;
SQLSMALLINT pcco=0;

SQLINTEGER SPCODE;
struct {
SQLSMALLINT LEN;
SQLCHAR DATA[200]; } STMTSQL;

```

Example code

```
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(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 */
SQLINTEGER
main()
{
    printf("\nAPD29 INITIALIZATION");
    scale = 0;
    rc=0;

    printf("\nAPD29 SQLAllocHandle-Environment stmt=%d",__LINE__);
    henv=0;
    rc = SQLAllocHandle(SQL_HANDLE_ENV, &henv, rc);
    CHECK_HANDLE (SQL_HANDLE_ENV, henv, rc);
    printf("\nAPD29-henv=%i",henv);

for (locix=1;locix<=2;locix++)
{
    /* Start SQL statements */
    printf("\nAPD29 SQLAllocHandle-Connection ");
    hdbc=0;
    SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc);
    CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);
    printf("\nAPD29-hdbc=%i",hdbc);
    printf("\nAPD29 SQLConnect ");
    if (locix == 1)
    {
        strcpy((char *)server,(char *)locsite);
    }
    else
    {
        strcpy((char *)server,(char *)remsite);
    }
}
}

```

Example code

```
strcpy((char *)uid,"cliuser");
strcpy((char *)pwd,"password");
printf("\nAPD29 server=%s",server);
printf("\nAPD29 uid=%s",uid);
printf("\nAPD29 pwd=%s",pwd);
rc=SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);
/*****
/* CASE(SPCODE=0) QRS RETURNED=0 COL=0 ROW=0 */
*****/
printf("\nAPD29 SQLAllocHandle-Statement stmt=%d",__LINE__);
hstmt=0;
rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
printf("\nAPD29-hstmt=%i\n",hstmt);

SPCODE=0;
printf("\nAPD29 call sp SPCODE =%i\n",SPCODE);
printf("\nAPD29 SQLPrepare stmt=%d",__LINE__);
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=%s",sqlstmt);
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nAPD29 SQLBindParameter stmt=%d",__LINE__);
rc = SQLBindParameter(hstmt,
                      1,
                      SQL_PARAM_INPUT_OUTPUT,
                      SQL_C_LONG,
                      SQL_INTEGER,
                      0,
                      0,
                      &SPCODE,
                      0,
                      NULL);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nAPD29 SQLExecute stmt=%d",__LINE__);
rc=SQLExecute(hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
{
    printf("\nAPD29 SPCODE not zero, spcode=%i\n",SPCODE);
    goto dberror;
}

printf("\nAPD29 SQLEndTran stmt=%d",__LINE__);
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);

printf("\nAPD29 SQLFreeHandle-Statement stmt=%d",__LINE__);
rc=SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
/*****
/* CASE(SPCODE=1) QRS RETURNED=0 COL=0 ROW=0 */
*****/
printf("\nAPD29 SQLAllocHandle-Statement stmt=%d",__LINE__);
hstmt=0;
rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
printf("\nAPD29-hstmt=%i\n",hstmt);
```

```

SPCODE=1;
printf("\nAPD29 call sp SPCODE =%i\n",SPCODE);
printf("\nAPD29 SQLPrepare stmt=%d",__LINE__);
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=%s",sqlstmt);
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nAPD29 SQLBindParameter stmt=%d",__LINE__);
rc = SQLBindParameter(hstmt,
                      1,
                      SQL_PARAM_INPUT_OUTPUT,
                      SQL_C_LONG,
                      SQL_INTEGER,
                      0,
                      0,
                      &SPCODE,
                      0,
                      NULL);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nAPD29 SQLExecute stmt=%d",__LINE__);
rc=SQLExecute(hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
if( SPCODE != 0 )
{
    printf("\nAPD29 SPCODE not zero, spcode=%i\n",SPCODE);
    goto dberror;
}

printf("\nAPD29 SQLEndTran stmt=%d",__LINE__);
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);

printf("\nAPD29 SQLFreeHandle-Statement stmt=%d",__LINE__);
rc=SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
/*****
/* CASE(SPCODE=2) QRS RETURNED=2 COL=1(int4/chr10) ROW=100+ */
*****/
printf("\nAPD29 SQLAllocHandle-Statement number=18\n");
hstmt=0;
rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
printf("\nAPD29-hstmt=%i\n",hstmt);

SPCODE=2;
printf("\nAPD29 call sp SPCODE =%i\n",SPCODE);
printf("\nAPD29 SQLPrepare          number= 19\n");
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=%s",sqlstmt);
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

```

Example code

```
printf("\nAPD29 SQLBindParameter      number= 20\n");
rc = SQLBindParameter(hstmt,
                      1,
                      SQL_PARAM_INPUT_OUTPUT,
                      SQL_C_LONG,
                      SQL_INTEGER,
                      0,
                      0,
                      &SPCODE,
                      0,
                      NULL);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nAPD29 SQLExecute            number= 21\n");
rc=SQLExecute(hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
if( SPCODE != 0 )
{
    printf("\nAPD29 socode incorrect");
    goto dberror;
}

printf("\nAPD29 SQLNumResultCols      number= 22\n");
rc=SQLNumResultCols(hstmt,&pccol);
if (pccol!=1)
{
    printf("APD29 col count wrong=%i\n",pccol);
    goto dberror;
}

printf("\nAPD29 SQLBindCol            number= 23\n");
rc=SQLBindCol(hstmt,
              1,
              SQL_C_LONG,
              (SQLPOINTER) &H1INT4,
              (SQLINTEGER)sizeof(SQLINTEGER),
              (SQLINTEGER *) &LNH1INT4 );
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

jx=0;
printf("\nAPD29 SQLFetch              number= 24\n");
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
{
    jx++;
    printf("\nAPD29 fetch loop jx =%i\n",jx);
    if ( ( H1INT4<=0) || ( H1INT4>=202)
        || (LNH1INT4!=4 && LNH1INT4!--1) )
    { /* data error */
        printf("\nAPD29 H1INT4=%i\n",H1INT4);
        printf("\nAPD29 LNH1INT4=%i\n",LNH1INT4);
        goto dberror;
    }
    printf("\nAPD29 SQLFetch          number= 24\n");
} /* end while loop */

if( rc != SQL_NO_DATA_FOUND )
{
    printf("\nAPD29 invalid end of data\n");
    check_error(SQL_HANDLE_STMT,hstmt,rc,__LINE__,__FILE__);
    goto dberror;
}
```

```

printf("\nAPD29 SQLMoreResults      number= 25\n");
rc=SQLMoreResults(hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nAPD29 SQLNumResultCols    number= 26\n");
rc=SQLNumResultCols(hstmt,&pccol);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
if (pccol!=1) {
    printf("APD29 col count wrong=%i\n",pccol);
    goto dberror;
}

printf("\nAPD29 SQLBindCol          number= 27\n");
rc=SQLBindCol(hstmt,
              1,
              SQL_C_CHAR,
              (SQLPOINTER) H1CHR10,
              (SQLINTEGER)sizeof(H1CHR10),
              (SQLINTEGER *) &LNH1CHR10
              );
if( rc != SQL_SUCCESS ) goto dberror;

jx=0;
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
{
    jx++;
    printf("\nAPD29 fetch loop jx =%i\n",jx);
    result=strcmp((char *)H1CHR10,"CHAR      ");
    if ( (result!=0)
        || (LNH1INT4!=4 && LNH1INT4!=-1) )
    {
        printf("\nAPD29 H1CHR10=%s\n",H1CHR10);
        printf("\nAPD29 result=%i\n",result);
        printf("\nAPD29 LNH1CHR10=%i\n",LNH1CHR10);
        printf("\nAPD29 strlen(H1CHR10)=%i\n",strlen((char *)H1CHR10));
        goto dberror;
    }
    printf("\nAPD29 SQLFetch          number= 24\n");
} /* end while loop */

if( rc != SQL_NO_DATA_FOUND )
check_error(SQL_HANDLE_STMT,hstmt,rc,__LINE__,__FILE__);
goto dberror;

printf("\nAPD29 SQLMoreResults      number= 29\n");
rc=SQLMoreResults(hstmt);
if(rc !=SQL_NO_DATA_FOUND ){
check_error(SQL_HANDLE_STMT,hstmt,rc,__LINE__,__FILE__);
goto dberror;
}

printf("\nAPD29 SQLEndTran number=30 \n");
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);

printf("\nAPD29 SQLFreeHandle-Statement number=31");
rc=SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
CHECK_HANDLE (SQL_HANDLE_DBC, hstmt, rc);
/*****
    printf("\nAPD29 SQLDisconnect stmt=%d",__LINE__);
    rc=SQLDisconnect(hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;
*****/
printf("\nSQLFreeHandle-Connection stmt=%d",__LINE__);
rc=SQLFreeHandle (SQL_HANDLE_DBC, hdbc);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);
/*****
/* End SQL statements *****/

```

Example code

```
    } /* end for each site perform these stmts */

    for (locix=1;locix<=2;locix++)
    {
        /*****
        printf("\nAPD29 SQLAllocHandle-Connection ");
        hdbc=0;
        rc=SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc);
        CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);
        printf("\nAPD29-hdbc=%i",hdbc);
        *****/
        printf("\nAPD29 SQLConnect                               ");
        if (locix == 1)
        {
            strcpy((char *)server,(char *)locsite);
        }
        else
        {
            strcpy((char *)server,(char *)remsite);
        }

        strcpy((char *)uid,"cliuser");
        strcpy((char *)pwd,"password");
        printf("\nAPD29 server=%s",server);
        printf("\nAPD29 uid=%s",uid);
        printf("\nAPD29 pwd=%s",pwd);
        rc=SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
        if( rc != SQL_SUCCESS ) goto dberror;
        /*****
        /* Start validate SQL statements *****/
        *****/
        printf("\nAPD01 SQLAllocHandle-Statement \n");
        hstmt=0;
        rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
        CHECK_HANDLE (SQL_HANDLE_DBC, hstmt, rc);
        printf("\nAPD01-hstmt=%i\n",hstmt);

        printf("\nAPD01 SQLExecDirect                               \n");
        strcpy((char *)sqlstmt,
        "SELECT INT4 FROM TABLE2A WHERE INT4=200");
        printf("\nAPD01 sqlstmt=%s",sqlstmt);
        rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
        CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

        printf("\nAPD01 SQLBindCol                               \n");
        rc=SQLBindCol(hstmt,
            1,
            SQL_C_LONG,
            (SQLPOINTER) &H1INT4,
            (SQLINTEGER)sizeof(SQLINTEGER),
            (SQLINTEGER *) &LNH1INT4 );
        CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

        printf("\nAPD01 SQLFetch                               \n");
        rc=SQLFetch(hstmt);
        CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
        if ((H1INT4!=200) || (LNH1INT4!=4))
        {
            printf("\nAPD01 H1INT4=%i\n",H1INT4);
            printf("\nAPD01 LNH1INT4=%i\n",LNH1INT4);
            goto dberror;
        }
    }
}
```

```

printf("\nAPD01 SQLEndTran \n");
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);

printf("\nAPD01 SQLCloseCursor \n");
rc=SQLCloseCursor (hstmt);
CHECK_HANDLE (SQL_HANDLE_DBC, hstmt, rc);

printf("\nAPD01 SQLExecDirect \n");
strcpy((char *)sqlstmt,
"SELECT INT4 FROM TABLE2A WHERE INT4=201");
printf("\nAPD01 sqlstmt=%s",sqlstmt);
rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);

printf("\nAPD01 SQLFetch \n");
rc=SQLFetch(hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
if ((H1INT4!=201) || (LNH1INT4!=4))
{
printf("\nAPD01 H1INT4=%i\n",H1INT4);
printf("\nAPD01 LNH1INT4=%i\n",LNH1INT4);
goto dberror;
}

printf("\nAPD01 SQLEndTran \n");
rc=SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);

printf("\nAPD01 SQLFreeHandle-Statement \n");
rc=SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
CHECK_HANDLE (SQL_HANDLE_STMT, hstmt, rc);
/*****
/* End validate SQL statements *****/
/*****
printf("\nAPD29 SQLDisconnect stmt=%d",__LINE__);
rc=SQLDisconnect(hdbc);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);
/*****
printf("\nSQLFreeHandle-Connection stmt=%d",__LINE__);
rc=SQLFreeHandle (SQL_HANDLE_DBC, hdbc);
CHECK_HANDLE (SQL_HANDLE_DBC, hdbc, rc);

} /* end for each site perform these stmts */
/*****
printf("\nSQLFreeHandle-Environment stmt=%d",__LINE__);
rc=SQLFreeHandle (SQL_HANDLE_ENV, henv);
CHECK_HANDLE (SQL_HANDLE_ENV, henv, rc);
/*****
goto pgmend;

dberror:
printf("\nAPD29 entry dberror label");
printf("\nAPD29 rc=%d",rc);
printf("\nAPDXX SQLFreeHandle-Environment number=6 \n");
rc=SQLFreeHandle (SQL_HANDLE_ENV, henv);
printf("\nAPDXX FREEENV rc =%d",rc);
rc=12;
printf("\nAPDXX DBERROR set rc =%d",rc);
goto pgmend;

```

Example code

```
pgmend:
printf("\nAPD29 TERMINATION ");
if (rc!=0)
{
printf("\nAPD29 WAS NOT SUCCESSFUL");
printf("\nAPD29 SPCODE = %i", SPCODE );
printf("\nAPD29 rc = %i", rc );
}
else
printf("\nAPD29 WAS SUCCESSFUL");

return(rc);

} /*END MAIN*/
/*****
** check_error - call print_error(), checks severity of return code
*****/
SQLRETURN check_error( 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("\nSPD29 entry check_error rtn");

switch (frc){
case SQL_SUCCESS:
break;
case SQL_INVALID_HANDLE:
printf("\nSPD29 check_error>SQL_INVALID_HANDLE ");
break;
case SQL_ERROR:
printf("\nSPD29 check_error>SQL_ERROR ");
break;
case SQL_SUCCESS_WITH_INFO:
printf("\nSPD29 check_error>SQL_SUCCESS_WITH_INFO");
break;
case SQL_NO_DATA_FOUND:
printf("\nSPD29 check_error>SQL_NO_DATA_FOUND ");
break;
default:
printf("\nSPD29 check_error>Invalid rc from api rc=%i",frc);
break;
}/*end switch*/
```

```

print_error(htype,hndl,frc,line,file);
printf("\nSPD29 SQLGetSQLCA ");
rc =SQLGetSQLCA(henv,hdbc,hstmt,&sqlca);
if(rc ==SQL_SUCCESS )
    prt_sqlca();
else
    printf("\n SPD29-check_error SQLGetSQLCA failed rc=%i",rc);
return (frc);
}
/*****
/* print_error
/* calls SQLGetDiagRec() and 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("\nSPD29 entry print_error rtn");
    printf("\nSPD29 rc=%d reported fro file:%s,line:%d --- ",
           frc,
           file,
           line
           );
    i = 1;
    while (SQLGetDiagRec( htype,
                        hndl,
                        i,
                        sqlstate,
                        &sqlcode,
                        buffer,
                        SQL_MAX_MESSAGE_LENGTH + 1,
                        &length
                        ) == SQL_SUCCESS ){
        printf("\nSPD29 SQLSTATE: %s", sqlstate);
        printf("\nSPD29 Native Error Code: %ld", sqlcode);
        printf("\nSPD29 buffer: %s", buffer);
        i++;
    }
    printf("\n>-----");
    printf("\nSPD29 exit print_error rtn");
    return(SQL_ERROR);
}
/*****
/*          P r i n t   S Q L C A
*****/
SQLRETURN
prt_sqlca()
{
    SQLINTEGER i;
    printf("\nIAPD29 entry prts_sqlca rtn");
    printf("\r\r*** Printing the SQLCA:\r");
    printf("\nSQLCAID   .... %s",sqlca.sqlcaid);
    printf("\nSQLCABC   .... %d",sqlca.sqlcabc);
    printf("\nSQLCODE   .... %d",sqlca.sqlcode);
    printf("\nSQLERRML  ... %d",sqlca.sqlerrml);
    printf("\nSQLERRMC  ... %s",sqlca.sqlerrmc);
    printf("\nSQLERRP   ... %s",sqlca.sqlerrp);
    for (i = 0; i < 6; i++)
        printf("\nSQLERRD%d ... %d",i+1,sqlca.sqlerrd??(i??));
    for (i = 0; i < 10; i++)
        printf("\nSQLWARN%d ... %c",i,sqlca.sqlwarn[i]);
    printf("\nSQLWARNA ... %c",sqlca.sqlwarn[10]);
    printf("\nSQLSTATE ... %s",sqlca.sqlstate);
}

```

Appendix G. Notices

This information was developed for products and services offered in the U.S.A. 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
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs

and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

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 the names and addresses used by an actual business enterprise 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.

Programming interface information

This book is intended to help the customer write applications that use DB2 ODBC to access IBM DB2 for OS/390 and z/OS servers. This book documents General-use Programming Interface and Associated Guidance Information provided by DATABASE 2 for OS/390 (DB2 for OS/390).

General-use programming interfaces allow the customer to write programs that obtain the services of DB2 for OS/390.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both.

AD/Cycle	ES/3090
APL2	IBM
AS/400	IBM Registry
BookManager	IMS
C/370	IMS/ESA
CICS	Language Environment
CICS/ESA	MVS/DFP
DATABASE 2	MVS/ESA
DataHub	Net.Data
DataPropagator	OpenEdition
DB2	OS/2
DB2 Connect	OS/390
DB2 Universal Database	OS/400
DFSMSdfp	Parallel Sysplex
DFSMSdss	QMF
DFSMSHsm	RACF
DFSMS/MVS	RAMAC
DFSORT	SAA
Distributed Relational Database Architecture	SecureWay
DRDA	SQL/DS
Enterprise Storage Server	System/370
Enterprise System/3090	System/390
Enterprise System/9000	VTAM
	WebSphere
	z/OS

#

Tivoli and NetView are trademarks of Tivoli Systems, Inc. in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are registered 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.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

The following terms and abbreviations are defined as they are used in the DB2 library.

A

address space. A range of virtual storage pages that is identified by a number (ASID) and a collection of segment and page tables that map the virtual pages to real pages of the computer's memory.

address space connection. The result of connecting an allied address space to DB2. Each address space that contains a task that is connected to DB2 has exactly one address space connection, even though more than one task control block (TCB) can be present. See also *allied address space* and *task control block*.

allied address space. An area of storage that is external to DB2 and that is connected to DB2. An allied address space is capable of requesting DB2 services.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

ANSI. American National Standards Institute.

API. Application programming interface.

application. A program or set of programs that performs a task; for example, a payroll application.

| **application-directed connection.** A connection that
| an application manages using the SQL CONNECT
| statement.

application plan. The control structure that is produced during the bind process. DB2 uses the application plan to process SQL statements that it encounters during statement execution.

application programming interface (API). A functional interface that is supplied by the operating system or by a separately orderable licensed program that allows an application program that is written in a high-level language to use specific data or functions of the operating system or licensed program.

| **application requester.** The component on a remote
| system that generates DRDA requests for data on
| behalf of an application. An application requester
| accesses a DB2 database server using the DRDA
| application-directed protocol.

| **ASCII.** An encoding scheme that is used to represent
| strings in many environments, typically on PCs and
| workstations. Contrast with *EBCDIC* and *Unicode*.

attachment facility. An interface between DB2 and TSO, IMS, CICS, or batch address spaces. An attachment facility allows application programs to access DB2.

authorization ID. A string that can be verified for connection to DB2 and to which a set of privileges is allowed. It can represent an individual, an organizational group, or a function, but DB2 does not determine this representation.

B

base table. (1) A table that is created by the SQL CREATE TABLE statement and that holds persistent data. Contrast with *result table* and *temporary table*.

(2) A table containing a LOB column definition. The actual LOB column data is not stored with the base table. The base table contains a row identifier for each row and an indicator column for each of its LOB columns. Contrast with *auxiliary table*.

binary large object (BLOB). A sequence of bytes where the size of the value ranges from 0 bytes to 2 GB-1. Such a string does not have an associated CCSID.

bind. The process by which the output from the SQL precompiler is converted to a usable control structure, often called an access plan, application plan, or package. During this process, access paths to the data are selected and some authorization checking is performed. The types of bind are:

automatic bind. (More correctly, *automatic rebind*) A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.

dynamic bind. A process by which SQL statements are bound as they are entered.

incremental bind. A process by which SQL statements are bound during the execution of an application process, because they could not be bound during the bind process, and VALIDATE(RUN) was specified.

static bind. A process by which SQL statements are bound after they have been precompiled. All static SQL statements are prepared for execution at the same time.

BLOB. Binary large object.

built-in function. A function that DB2 supplies. Contrast with *user-defined function*.

C

CAF. Call attachment facility.

call attachment facility (CAF). A DB2 attachment facility for application programs that run in TSO or MVS batch. The CAF is an alternative to the DSN command processor and provides greater control over the execution environment.

call level interface (CLI). A callable application programming interface (API) for database access, which is an alternative to using embedded SQL. In contrast to embedded SQL, DB2 ODBC (which is based on the CLI architecture) does not require the user to precompile or bind applications, but instead provides a standard set of functions to process SQL statements and related services at run time.

cast function. A function that is used to convert instances of a (source) data type into instances of a different (target) data type. In general, a cast function has the name of the target data type. It has one single argument whose type is the source data type; its return type is the target data type.

catalog. In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

catalog table. Any table in the DB2 catalog.

character large object (CLOB). A sequence of bytes representing single-byte characters or a mixture of single- and double-byte characters where the size of the value can be up to 2 GB–1. In general, character large object values are used whenever a character string might exceed the limits of the VARCHAR type.

CLI. Call level interface.

client. See *requester*.

CLOB. Character large object.

column function. An operation that derives its result by using values from one or more rows. Contrast with *scalar function*.

commit. The operation that ends a unit of work by releasing locks so that the database changes that are made by that unit of work can be perceived by other processes.

connection handle. The data object containing information that is associated with a connection that is managed by DB2 ODBC. This includes general status information, transaction status, and diagnostic information.

constant. A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. Contrast with *variable*.

context. The application's logical connection to the data source and associated internal DB2 ODBC connection information that allows the application to direct its operations to a data source. A DB2 ODBC context represents a DB2 thread.

cursor. A named control structure that an application program uses to point to a row of interest within some set of rows, and to retrieve rows from the set, possibly making updates or deletions.

D

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and the access to the data stored within it.

data source. A local or remote relational or non-relational data manager that is capable of supporting data access via an ODBC driver that supports the ODBC APIs. In the case of DB2 for OS/390 and z/OS, the data sources are always relational database managers.

DBCLOB. Double-byte character large object.

DBMS. Database management system.

DB2 for VSE & VM. The IBM DB2 relational database management system for the VSE and VM operating systems.

DB2 thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services.

dimension. A data category such as time, products, or markets. The elements of a dimension are referred to as members. Dimensions offer a very concise, intuitive way of organizing and selecting data for retrieval, exploration, and analysis. See also *dimension table*.

dimension table. The representation of a dimension in a star schema. Each row in a dimension table represents all of the attributes for a particular member of the dimension. See also *dimension*, *star schema*, and *star join*.

distinct type. A user-defined data type that is internally represented as an existing type (its source type), but is considered to be a separate and incompatible type for semantic purposes.

Distributed Relational Database Architecture (DRDA). A connection protocol for distributed relational database processing that is used by IBM's relational database products. DRDA includes protocols for communication between an application and a remote

relational database management system, and for communication between relational database management systems.

double-byte character large object (DBCLOB). A sequence of bytes representing double-byte characters where the size of the values can be up to 2 GB. In general, double-byte character large object values are used whenever a double-byte character string might exceed the limits of the VARGRAPHIC type.

DRDA. Distributed Relational Database Architecture.

dynamic SQL. SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution.

E

EBCDIC. Extended binary coded decimal interchange code. An encoding scheme that is used to represent character data in the OS/390, MVS, VM, VSE, and OS/400 environments. Contrast with *ASCII* and *Unicode*.

embedded SQL. SQL statements that are coded within an application program. See *static SQL*.

enclave. In Language Environment, an independent collection of routines, one of which is designated as the main routine. An enclave is similar to a program or run unit.

environment. A collection of names of logical and physical resources that are used to support the performance of a function.

environment handle. In DB2 ODBC, the data object that contains global information regarding the state of the application. An environment handle must be allocated before a connection handle can be allocated. Only one environment handle can be allocated per application.

equijoin. A join operation in which the join-condition has the form *expression = expression*.

external function. A function for which the body is written in a programming language that takes scalar argument values and produces a scalar result for each invocation. Contrast with *sourced function*, *built-in function*, and *SQL function*.

F

foreign key. A column or set of columns in a dependent table of a constraint relationship. The key must have the same number of columns, with the same

descriptions, as the primary key of the parent table. Each foreign key value must either match a parent key value in the related parent table or be null.

full outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of both tables. See also *join*.

function. A mapping, embodied as a program (the function body), invocable by means of zero or more input values (arguments), to a single value (the result). See also *column function* and *scalar function*.

Functions can be user-defined, built-in, or generated by DB2. (See *built-in function*, *cast function*, *external function*, *sourced function*, *SQL function*, and *user-defined function*.)

H

handle. In DB2 ODBC, a variable that refers to a data structure and associated resources. See *statement handle*, *connection handle*, and *environment handle*.

I

initialization file. For DB2 ODBC applications, a file containing values that can be set to adjust the performance of the database manager.

inner join. The result of a join operation that includes only the matched rows of both tables being joined. See also *join*.

L

large object (LOB). A sequence of bytes representing bit data, single-byte characters, double-byte characters, or a mixture of single- and double-byte characters. A LOB can be up to 2 GB–1 byte in length. See also *BLOB*, *CLOB*, and *DBCLOB*.

left outer join. The result of a join operation that includes the matched rows of both tables that are being joined, and that preserves the unmatched rows of the first table. See also *join*.

linkage editor. A computer program for creating load modules from one or more object modules or load modules by resolving cross references among the modules and, if necessary, adjusting addresses.

link-edit. The action of creating a loadable computer program using a linkage editor.

load module. A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

LOB. Large object.

LOB locator • reentrant

LOB locator. A mechanism that allows an application program to manipulate a large object value in the database system. A LOB locator is a fullword integer value that represents a single LOB value. An application program retrieves a LOB locator into a host variable and can then apply SQL operations to the associated LOB value using the locator.

local. A way of referring to any object that the local DB2 subsystem maintains. A *local table*, for example, is a table that is maintained by the local DB2 subsystem. Contrast with *remote*.

M

multithreading. Multiple TCBs that are executing one copy of DB2 ODBC code concurrently (sharing a processor) or in parallel (on separate central processors).

mutex. Pthread mutual exclusion; a lock. A Pthread mutex variable is used as a locking mechanism to allow serialization of critical sections of code by temporarily blocking the execution of all but one thread.

MVS/ESA™. Multiple Virtual Storage/Enterprise Systems Architecture.

N

NUL. In C, a single character that denotes the end of the string.

null. A special value that indicates the absence of information.

NUL-terminated host variable. A varying-length host variable in which the end of the data is indicated by the presence of a NUL terminator.

NUL terminator. In C, the value that indicates the end of a string. For character strings, the NUL terminator is X'00'.

O

ODBC. Open Database Connectivity.

ODBC driver. A dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

Open Database Connectivity (ODBC). A Microsoft database application programming interface (API) for C that allows access to database management systems by using callable SQL. ODBC does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that lets users add modules called *database drivers*, which link the application to their choice of database management systems at run time. This means that applications no longer need to be

directly linked to the modules of all the database management systems that are supported.

OS/390. Operating System/390®.

outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves some or all of the unmatched rows of the tables that are being joined. See also *join*.

P

plan. See *application plan*.

plan name. The name of an application plan.

POSIX. Portable Operating System Interface. The IEEE operating system interface standard, which defines the Pthread standard of threading. See *Pthread*.

precompilation. A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.

prepare. The first phase of a two-phase commit process in which all participants are requested to prepare for commit.

prepared SQL statement. A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.

primary key. In a relational database, a unique, nonnull key that is part of the definition of a table. A table cannot be defined as a parent unless it has a unique key or primary key.

Pthread. The POSIX threading standard model for splitting an application into subtasks. The Pthread standard includes functions for creating threads, terminating threads, synchronizing threads through locking, and other thread control facilities.

R

RDBMS. Relational database management system.

reentrant. Executable code that can reside in storage as one shared copy for all threads. Reentrant code is not self-modifying and provides separate storage areas for each thread. Reentrancy is a compiler and operating system concept, and reentrancy alone is not enough to guarantee logically consistent results when multithreading. See *threadsafe*.

relational database management system (RDBMS).

A collection of hardware and software that organizes and provides access to a relational database.

remote. Any object that is maintained by a remote DB2 subsystem (that is, by a DB2 subsystem other than the local one). A *remote view*, for example, is a view that is maintained by a remote DB2 subsystem. Contrast with *local*.

| **requester.** The source of a request to access data at a remote server. In the DB2 environment, the requester function is provided by the distributed data facility.

result set. The set of rows that a stored procedure returns to a client application.

result set locator. A 4-byte value that DB2 uses to uniquely identify a query result set that a stored procedure returns.

result table. The set of rows that are specified by a SELECT statement.

right outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of the second join operand. See also *join*.

rollback. The process of restoring data changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with *commit*.

S

scalar function. An SQL operation that produces a single value from another value and is expressed as a function name, followed by a list of arguments that are enclosed in parentheses. Contrast with *column function*.

sourced function. A function that is implemented by another built-in or user-defined function that is already known to the database manager. This function can be a scalar function or a column (aggregating) function; it returns a single value from a set of values (for example, MAX or AVG). Contrast with *built-in function*, *external function*, and *SQL function*.

source type. An existing type that is used to internally represent a distinct type.

SQL. Structured Query Language.

SQL authorization ID (SQL ID). The authorization ID that is used for checking dynamic SQL statements in some situations.

SQLCA. SQL communication area.

SQL communication area (SQLCA). A structure that is used to provide an application program with information about the execution of its SQL statements.

SQLDA. SQL descriptor area.

SQL descriptor area (SQLDA). A structure that describes input variables, output variables, or the columns of a result table.

SQL/DS. Structured Query Language/Data System. This product is now obsolete and has been replaced by DB2 for VSE & VM.

SQL function. A user-defined function in which the CREATE FUNCTION statement contains the source code. The source code is a single SQL expression that evaluates to a single value. The SQL user-defined function can return only one parameter.

star join. A method of joining a dimension column of a fact table to the key column of the corresponding dimension table. See also *join*, *dimension*, and *star schema*.

star schema. The combination of a fact table (which contains most of the data) and a number of dimension tables. See also *star join*, *dimension*, and *dimension table*.

statement handle. In DB2 ODBC, the data object that contains information about an SQL statement that is managed by DB2 ODBC. This includes information such as dynamic arguments, bindings for dynamic arguments and columns, cursor information, result values, and status information. Each statement handle is associated with the connection handle.

static SQL. SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of host variables that are specified by the statement might change).

stored procedure. A user-written application program that can be invoked through the use of the SQL CALL statement.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

| **system-directed connection.** A connection that an RDBMS manages by processing SQL statements with three-part names.

T

table. A named data object consisting of a specific number of columns and some number of unordered rows. See also *base table* or *temporary table*.

task control block (TCB). A control block that is used to communicate information about tasks within an address space that are connected to DB2. An address space can support many task connections (as many as

TCB • z/OS

one per task), but only one address space connection. See also *address space connection*.

TCB. Task control block (in MVS).

TCP/IP. A network communication protocol that computer systems use to exchange information across telecommunication links.

temporary table. A table that holds temporary data; for example, temporary tables are useful for holding or sorting intermediate results from queries that contain a large number of rows. The two kinds of temporary table, which are created by different SQL statements, are the created temporary table and the declared temporary table. Contrast with *result table*. See also *created temporary table* and *declared temporary table*.

threadsafe. A characteristic of code that allows multithreading both by providing private storage areas for each thread, and by properly serializing shared (global) storage areas.

timestamp. A seven-part value that consists of a date and time. The timestamp is expressed in years, months, days, hours, minutes, seconds, and microseconds.

trace. A DB2 facility that provides the ability to monitor and collect DB2 monitoring, auditing, performance, accounting, statistics, and serviceability (global) data.

U

UDF. User-defined function.

UDT. User-defined data type. In DB2 for OS/390 and z/OS, the term *distinct type* is used instead of user-defined data type. See *distinct type*.

user-defined data type (UDT). See *distinct type*.

user-defined function (UDF). A function that is defined to DB2 by using the CREATE FUNCTION statement and that can be referenced thereafter in SQL statements. A user-defined function can be an *external function*, a *sourced function*, or an *SQL function*. Contrast with *built-in function*.

V

variable. A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a variable. Contrast with *constant*.

X

X/Open. An independent, worldwide open systems organization that is supported by most of the world's largest information systems suppliers, user organizations, and software companies. X/Open's goal

is to increase the portability of applications by combining existing and emerging standards.

Z

z/OS. An operating system for the eServer product line that supports 64-bit real storage.

Bibliography

DB2 Universal Database Server for OS/390 and z/OS Version 7 product libraries:

DB2 for OS/390 and z/OS

- *An Introduction to DB2 for OS/390*, SC26-9937
- *DB2 Administration Guide*, SC26-9931
- *DB2 Application Programming and SQL Guide*, SC26-9933
- *DB2 Application Programming Guide and Reference for Java*, SC26-9932
- *DB2 Command Reference*, SC26-9934
- *DB2 Data Sharing: Planning and Administration*, SC26-9935
- *DB2 Data Sharing Quick Reference Card*, SX26-3846
- *DB2 Diagnosis Guide and Reference*, LY37-3740
- *DB2 Diagnostic Quick Reference Card*, LY37-3741
- *DB2 Image, Audio, and Video Extenders Administration and Programming*, SC26-9947
- *DB2 Installation Guide*, GC26-9936
- *DB2 Licensed Program Specifications*, GC26-9938
- *DB2 Master Index*, SC26-9939
- *DB2 Messages and Codes*, GC26-9940
- *DB2 ODBC Guide and Reference*, SC26-9941
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC26-9942
- *DB2 Reference Summary*, SX26-3847
- *DB2 Release Planning Guide*, SC26-9943
- *DB2 SQL Reference*, SC26-9944
- *DB2 Text Extender Administration and Programming*, SC26-9948
- *DB2 Utility Guide and Reference*, SC26-9945
- *DB2 What's New?* GC26-9946
- *DB2 XML Extender for OS/390 and z/OS Administration and Programming*, SC27-9949
- *DB2 Program Directory*, GI10-8182

DB2 Administration Tool

- *DB2 Administration Tool for OS/390 and z/OS User's Guide*, SC26-9847

DB2 Buffer Pool Tool

- *DB2 Buffer Pool Tool for OS/390 and z/OS User's Guide and Reference*, SC26-9306

DB2 DataPropagator™

- *DB2 UDB Replication Guide and Reference*, SC26-9920

Net.Data®

The following books are available at this Web site:
www.ibm.com/software/net.data/library.html

- *Net.Data Library: Administration and Programming Guide for OS/390 and z/OS*
- *Net.Data Library: Language Environment Interface Reference*
- *Net.Data Library: Messages and Codes*
- *Net.Data Library: Reference*

DB2 PM for OS/390

- *DB2 PM for OS/390 Batch User's Guide*, SC27-0857
- *DB2 PM for OS/390 Command Reference*, SC27-0855
- *DB2 PM for OS/390 Data Collector Application Programming Interface Guide*, SC27-0861
- *DB2 PM for OS/390 General Information*, GC27-0852
- *DB2 PM for OS/390 Installation and Customization*, SC27-0860
- *DB2 PM for OS/390 Messages*, SC27-0856
- *DB2 PM for OS/390 Online Monitor User's Guide*, SC27-0858
- *DB2 PM for OS/390 Report Reference Volume 1*, SC27-0853
- *DB2 PM for OS/390 Report Reference Volume 2*, SC27-0854
- *DB2 PM for OS/390 Using the Workstation Online Monitor*, SC27-0859
- *DB2 PM for OS/390 Program Directory*, GI10-8223

Query Management Facility (QMF™)

- *Query Management Facility: Developing QMF Applications*, SC26-9579
- *Query Management Facility: Getting Started with QMF on Windows*, SC26-9582
- *Query Management Facility: High Performance Option User's Guide for OS/390 and z/OS*, SC26-9581

- *Query Management Facility: Installing and Managing QMF on OS/390 and z/OS, GC26-9575*
- *Query Management Facility: Installing and Managing QMF on Windows, GC26-9583*
- *Query Management Facility: Introducing QMF, GC26-9576*
- *Query Management Facility: Messages and Codes, GC26-9580*
- *Query Management Facility: Reference, SC26-9577*
- *Query Management Facility: Using QMF, SC26-9578*

Ada/370

- *IBM Ada/370 Language Reference, SC09-1297*
- *IBM Ada/370 Programmer's Guide, SC09-1414*
- *IBM Ada/370 SQL Module Processor for DB2 Database Manager User's Guide, SC09-1450*

APL2®

- *APL2 Programming Guide, SH21-1072*
- *APL2 Programming: Language Reference, SH21-1061*
- *APL2 Programming: Using Structured Query Language (SQL), SH21-1057*

AS/400®

The following books are available at this Web site:
www.as400.ibm.com/infocenter

- *DB2 Universal Database for AS/400 Database Programming*
- *DB2 Universal Database for AS/400 Performance and Query Optimization*
- *DB2 Universal Database for AS/400 Distributed Data Management*
- *DB2 Universal Database for AS/400 Distributed Data Programming*
- *DB2 Universal Database for AS/400 SQL Programming Concepts*
- *DB2 Universal Database for AS/400 SQL Programming with Host Languages*
- *DB2 Universal Database for AS/400 SQL Reference*

BASIC

- *IBM BASIC/MVS Language Reference, GC26-4026*
- *IBM BASIC/MVS Programming Guide, SC26-4027*

BookManager® READ/MVS

- *BookManager READ/MVS V1R3: Installation Planning & Customization, SC38-2035*

SAA® AD/Cycle® C/370™

- *IBM SAA AD/Cycle C/370 Programming Guide, SC09-1841*
- *IBM SAA AD/Cycle C/370 Programming Guide for Language Environment/370, SC09-1840*
- *IBM SAA AD/Cycle C/370 User's Guide, SC09-1763*
- *SAA CPI C Reference, SC09-1308*

Character Data Representation Architecture

- *Character Data Representation Architecture Overview, GC09-2207*
- *Character Data Representation Architecture Reference and Registry, SC09-2190*

CICS/ESA

- *CICS/ESA Application Programming Guide, SC33-1169*
- *CICS External Interfaces Guide, SC33-1944*
- *CICS for MVS/ESA Application Programming Reference, SC33-1170*
- *CICS for MVS/ESA CICS-RACF Security Guide, SC33-1185*
- *CICS for MVS/ESA CICS-Supplied Transactions, SC33-1168*
- *CICS for MVS/ESA Customization Guide, SC33-1165*
- *CICS for MVS/ESA Data Areas, LY33-6083*
- *CICS for MVS/ESA Installation Guide, SC33-1163*
- *CICS for MVS/ESA Intercommunication Guide, SC33-1181*
- *CICS for MVS/ESA Messages and Codes, GC33-1177*
- *CICS for MVS/ESA Operations and Utilities Guide, SC33-1167*
- *CICS/ESA Performance Guide, SC33-1183*
- *CICS/ESA Problem Determination Guide, SC33-1176*
- *CICS for MVS/ESA Resource Definition Guide, SC33-1166*
- *CICS for MVS/ESA System Definition Guide, SC33-1164*
- *CICS for MVS/ESA System Programming Reference, GC33-1171*

CICS Transaction Server for OS/390

- *CICS Application Programming Guide, SC33-1687*
- *CICS External Interfaces Guide, SC33-1703*
- *CICS DB2 Guide, SC33-1939*
- *CICS Resource Definition Guide, SC33-1684*

IBM C/C++ for MVS/ESA

- *IBM C/C++ for MVS/ESA Library Reference, SC09-1995*
- *IBM C/C++ for MVS/ESA Programming Guide, SC09-1994*

IBM COBOL

- *IBM COBOL Language Reference, SC26-4769*
- *IBM COBOL for MVS & VM Programming Guide, SC26-9049*

Conversion Guide

- *IMS-DB and DB2 Migration and Coexistence Guide, GH21-1083*

Cooperative Development Environment

- *CoOperative Development Environment/370: Debug Tool, SC09-1623*

DataPropagator NonRelational

- *DataPropagator NonRelational MVS/ESA Administration Guide, SH19-5036*
- *DataPropagator NonRelational MVS/ESA Reference, SH19-5039*

Data Facility Data Set Services

- *Data Facility Data Set Services: User's Guide and Reference, SC26-4388*

Database Design

- *DB2 Design and Development Guide by Gabrielle Wiorkowski and David Kull, Addison Wesley, ISBN 0-20158-049-7*
- *Handbook of Relational Database Design by C. Fleming and B. Von Halle, Addison Wesley, ISBN 0-20111-434-8*

DataHub®

- *IBM DataHub General Information, GC26-4874*

Data Refresher

- *Data Refresher Relational Extract Manager for MVS GI10-9927*

DB2 Connect®

- *DB2 Connect Enterprise Edition for OS/2 and Windows: Quick Beginnings, GC09-2953*
- *DB2 Connect Enterprise Edition for UNIX: Quick Beginnings, GC09-2952*
- *DB2 Connect Personal Edition Quick Beginnings, GC09-2967*
- *DB2 Connect User's Guide, SC09-2954*

DB2 Red Books

- *DB2 UDB Server for OS/390 Version 6 Technical Update, SG24-6108-00*

DB2 Server for VSE & VM

- *DB2 Server for VM: DBS Utility, SC09-2394*
- *DB2 Server for VSE: DBS Utility, SC09-2395*

DB2 Universal Database for UNIX®, Windows®, OS/2®

- *DB2 UDB Administration Guide: Planning, SC09-2946*
- *DB2 UDB Administration Guide: Implementation, SC09-2944*
- *DB2 UDB Administration Guide: Performance, SC09-2945*
- *DB2 UDB Administrative API Reference, SC09-2947*
- *DB2 UDB Application Development Guide, Volume 3, SC09-2948*
- *DB2 UDB Application Development Guide, SC09-2949*
- *DB2 UDB CLI Guide and Reference, SC09-2950*
- *DB2 UDB Command Reference, SC09-2951*
- *DB2 UDB SQL Getting Started, SC09-2973*
- *DB2 UDB SQL Reference Volume 1, SC09-2974*
- *DB2 UDB SQL Reference Volume 2, SC09-2975*

Device Support Facilities

- *Device Support Facilities User's Guide and Reference, GC35-0033*

DFSMS

These books provide information about a variety of components of DFSMS, including DFSMS/MVS®, DFSMSdfp™, DFSMSdss™, DFSMSshsm™, and MVS/DFP™.

- *DFSMS/MVS: Access Method Services for the Integrated Catalog, SC26-4906*
- *DFSMS/MVS: Access Method Services for VSAM Catalogs, SC26-4905*
- *DFSMS/MVS: Administration Reference for DFSMSdss, SC26-4929*
- *DFSMS/MVS: DFSMSshsm Managing Your Own Data, SH21-1077*
- *DFSMS/MVS: Diagnosis Reference for DFSMSdfp, LY27-9606*
- *DFSMS/MVS Storage Management Library: Implementing System-Managed Storage, SC26-3123*
- *DFSMS/MVS: Macro Instructions for Data Sets, SC26-4913*
- *DFSMS/MVS: Managing Catalogs, SC26-4914*
- *DFSMS/MVS: Program Management, SC26-4916*

- *DFSMS/MVS: Storage Administration Reference for DFSMSdfp*, SC26-4920
- *DFSMS/MVS: Using Advanced Services*, SC26-4921
- *DFSMS/MVS: Utilities*, SC26-4926
- *OS/390 DFSMS: Using Data Sets*, SC26-4749

DFSORT™

- *DFSORT Application Programming: Guide*, SC33-4035

Distributed Relational Database Architecture™

- *Data Stream and OPA Reference*, SC31-6806
- *IBM SQL Reference*, SC26-8416
- *Open Group Technical Standard*

The Open Group presently makes the following DRDA books available through its Web site at: www.opengroup.org

- *DRDA Version 2 Vol. 1: Distributed Relational Database Architecture (DRDA)*
- *DRDA Version 2 Vol. 2: Formatted Data Object Content Architecture*
- *DRDA Version 2 Vol. 3: Distributed Data Management Architecture*

Domain Name System

- *DNS and BIND, Third Edition*, Paul Albitz and Cricket Liu, O'Reilly, ISBN 1-56592-512-2

Education

- *IBM Dictionary of Computing*, McGraw-Hill, ISBN 0-07031-489-6
- *1999 IBM All-in-One Education and Training Catalog*, GR23-8105

Enterprise System/9000® and Enterprise System/3090™

- *Enterprise System/9000 and Enterprise System/3090 Processor Resource/System Manager Planning Guide*, GA22-7123

High Level Assembler

- *High Level Assembler for MVS and VM and VSE Language Reference*, SC26-4940
- *High Level Assembler for MVS and VM and VSE Programmer's Guide*, SC26-4941

Parallel Sysplex® Library

- *OS/390 Parallel Sysplex Application Migration*, GC28-1863
- *System/390 MVS Sysplex Hardware and Software Migration*, GC28-1862
- *OS/390 Parallel Sysplex Overview: An Introduction to Data Sharing and Parallelism*, GC28-1860

- *OS/390 Parallel Sysplex Systems Management*, GC28-1861
- *OS/390 Parallel Sysplex Test Report*, GC28-1963
- *System/390 9672/9674 System Overview*, GA22-7148

ICSF/MVS

- *ICSF/MVS General Information*, GC23-0093

IMS

- *IMS Batch Terminal Simulator General Information*, GH20-5522
- *IMS Administration Guide: System*, SC26-9420
- *IMS Administration Guide: Transaction Manager*, SC26-9421
- *IMS Application Programming: Database Manager*, SC26-9422
- *IMS Application Programming: Design Guide*, SC26-9423
- *IMS Application Programming: Transaction Manager*, SC26-9425
- *IMS Command Reference*, SC26-9436
- *IMS Customization Guide*, SC26-9427
- *IMS Install Volume 1: Installation and Verification*, GC26-9429
- *IMS Install Volume 2: System Definition and Tailoring*, GC26-9430
- *IMS Messages and Codes*, GC27-1120
- *IMS Utilities Reference: System*, SC26-9441

ISPF

- *ISPF V4 Dialog Developer's Guide and Reference*, SC34-4486
- *ISPF V4 Messages and Codes*, SC34-4450
- *ISPF V4 Planning and Customizing*, SC34-4443
- *ISPF V4 User's Guide*, SC34-4484

Language Environment

- *Debug Tool User's Guide and Reference*, SC09-2137

MQSeries

- *MQSeries Application Messaging Interface*, SC34-5604
- *MQSeries for OS/390 Concepts and Planning Guide*, GC34-5650
- *MQSeries for OS/390 System Setup Guide*, SC34-5651

National Language Support

- *IBM National Language Support Reference Manual Volume 2*, SE09-8002

NetView®

- *NetView Installation and Administration Guide, SC31-8043*
- *NetView User's Guide, SC31-8056*

Microsoft ODBC

- *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference, Microsoft Press, ISBN 1-57231-516-4*

OS/390

- *OS/390 C/C++ Programming Guide, SC09-2362*
- *OS/390 C/C++ Run-Time Library Reference, SC28-1663*
- *OS/390 C/C++ User's Guide, SC09-2361*
- *OS/390 eNetwork Communications Server: IP Configuration, SC31-8513*
- *OS/390 Hardware Configuration Definition Planning, GC28-1750*
- *OS/390 Information Roadmap, GC28-1727*
- *OS/390 Introduction and Release Guide, GC28-1725*
- *OS/390 JES2 Initialization and Tuning Guide, SC28-1791*
- *OS/390 JES3 Initialization and Tuning Guide, SC28-1802*
- *OS/390 Language Environment for OS/390 & VM Concepts Guide, GC28-1945*
- *OS/390 Language Environment for OS/390 & VM Customization, SC28-1941*
- *OS/390 Language Environment for OS/390 & VM Debugging Guide, SC28-1942*
- *OS/390 Language Environment for OS/390 & VM Programming Guide, SC28-1939*
- *OS/390 Language Environment for OS/390 & VM Programming Reference, SC28-1940*
- *OS/390 MVS Diagnosis: Procedures, LY28-1082*
- *OS/390 MVS Diagnosis: Reference, SY28-1084*
- *OS/390 MVS Diagnosis: Tools and Service Aids, LY28-1085*
- *OS/390 MVS Initialization and Tuning Guide, SC28-1751*
- *OS/390 MVS Initialization and Tuning Reference, SC28-1752*
- *OS/390 MVS Installation Exits, SC28-1753*
- *OS/390 MVS JCL Reference, GC28-1757*
- *OS/390 MVS JCL User's Guide, GC28-1758*
- *OS/390 MVS Planning: Global Resource Serialization, GC28-1759*
- *OS/390 MVS Planning: Operations, GC28-1760*
- *OS/390 MVS Planning: Workload Management, GC28-1761*
- *OS/390 MVS Programming: Assembler Services Guide, GC28-1762*
- *OS/390 MVS Programming: Assembler Services Reference, GC28-1910*
- *OS/390 MVS Programming: Authorized Assembler Services Guide, GC28-1763*
- *OS/390 MVS Programming: Authorized Assembler Services Reference, Volumes 1-4, GC28-1764, GC28-1765, GC28-1766, GC28-1767*
- *OS/390 MVS Programming: Callable Services for High-Level Languages, GC28-1768*
- *OS/390 MVS Programming: Extended Addressability Guide, GC28-1769*
- *OS/390 MVS Programming: Sysplex Services Guide, GC28-1771*
- *OS/390 MVS Programming: Sysplex Services Reference, GC28-1772*
- *OS/390 MVS Programming: Workload Management Services, GC28-1773*
- *OS/390 MVS Routing and Descriptor Codes, GC28-1778*
- *OS/390 MVS Setting Up a Sysplex, GC28-1779*
- *OS/390 MVS System Codes, GC28-1780*
- *OS/390 MVS System Commands, GC28-1781*
- *OS/390 MVS System Messages Volume 1, GC28-1784*
- *OS/390 MVS System Messages Volume 2, GC28-1785*
- *OS/390 MVS System Messages Volume 3, GC28-1786*
- *OS/390 MVS System Messages Volume 4, GC28-1787*
- *OS/390 MVS System Messages Volume 5, GC28-1788*
- *OS/390 MVS Using the Subsystem Interface, SC28-1789*
- *OS/390 SecureWay Security Server Network Authentication and Privacy Service Administration, SC24-5896*
- *OS/390 Security Server External Security Interface (RACROUTE) Macro Reference, GC28-1922*
- *OS/390 Security Server (RACF) Auditor's Guide, SC28-1916*
- *OS/390 Security Server (RACF) Command Language Reference, SC28-1919*
- *OS/390 Security Server (RACF) General User's Guide, SC28-1917*
- *OS/390 Security Server (RACF) Introduction, GC28-1912*
- *OS/390 Security Server (RACF) Macros and Interfaces, SK2T-6700 (OS/390 Collection Kit), SK27-2180 (OS/390 Security Server Information Package)*
- *OS/390 Security Server (RACF) Security Administrator's Guide, SC28-1915*
- *OS/390 Security Server (RACF) System Programmer's Guide, SC28-1913*
- *OS/390 SMP/E Reference, SC28-1806*

- *OS/390 SMP/E User's Guide, SC28-1740*
- *OS/390 Support for Unicode: Using Conversion Services, SC33-7050*
- *OS/390 RMF User's Guide, SC28-1949*
- *OS/390 TSO/E CLISTS, SC28-1973*
- *OS/390 TSO/E Command Reference, SC28-1969*
- *OS/390 TSO/E Customization, SC28-1965*
- *OS/390 TSO/E Messages, GC28-1978*
- *OS/390 TSO/E Programming Guide, SC28-1970*
- *OS/390 TSO/E Programming Services, SC28-1971*
- *OS/390 TSO/E REXX Reference, SC28-1975*
- *OS/390 TSO/E User's Guide, SC28-1968*
- *OS/390 DCE Administration Guide, SC28-1584*
- *OS/390 DCE Introduction, GC28-1581*
- *OS/390 DCE Messages and Codes, SC28-1591*
- *OS/390 UNIX System Services Command Reference, SC28-1892*
- *OS/390 UNIX System Services Messages and Codes, SC28-1908*
- *OS/390 UNIX System Services Planning, SC28-1890*
- *OS/390 UNIX System Services User's Guide, SC28-1891*
- *OS/390 UNIX System Services Programming: Assembler Callable Services Reference, SC28-1899*
- *OS/390 V2R10.0 IBM CS IP Configuration Reference, SC31-8726*
- *Program Directory for OS/390 V2 R8/R9/R10 support for Unicode, GI10-9760*

IBM Enterprise PL/I for z/OS and OS/390

- *IBM Enterprise PL/I for z/OS and OS/390 Language Reference, SC26-9476*
- *IBM Enterprise PL/I for z/OS and OS/390 Programming Guide, SC26-9473*

OS PL/I

- *OS PL/I Programming Language Reference, SC26-4308*
- *OS PL/I Programming Guide, SC26-4307*

Prolog

- *IBM SAA AD/Cycle Prolog/MVS & VM Programmer's Guide, SH19-6892*

RAMAC® and Enterprise Storage Server™

- *IBM RAMAC Virtual Array, SG24-4951*
- *RAMAC Virtual Array: Implementing Peer-to-Peer Remote Copy, SG24-5338*
- *Enterprise Storage Server Introduction and Planning, GC26-7294*

Remote Recovery Data Facility

- *Remote Recovery Data Facility Program Description and Operations, LY37-3710*

Storage Management

- *DFSMS/MVS Storage Management Library: Implementing System-Managed Storage, SC26-3123*
- *MVS/ESA Storage Management Library: Leading a Storage Administration Group, SC26-3126*
- *MVS/ESA Storage Management Library: Managing Data, SC26-3124*
- *MVS/ESA Storage Management Library: Managing Storage Groups, SC26-3125*
- *MVS Storage Management Library: Storage Management Subsystem Migration Planning Guide, SC26-4659*

System/370™ and System/390

- *ESA/370 Principles of Operation, SA22-7200*
- *ESA/390 Principles of Operation, SA22-7201*
- *System/390 MVS Sysplex Hardware and Software Migration, GC28-1210*

System Network Architecture (SNA)

- *SNA Formats, GA27-3136*
- *SNA LU 6.2 Peer Protocols Reference, SC31-6808*
- *SNA Transaction Programmer's Reference Manual for LU Type 6.2, GC30-3084*
- *SNA/Management Services Alert Implementation Guide, GC31-6809*

TCP/IP

- *IBM TCP/IP for MVS: Customization & Administration Guide, SC31-7134*
- *IBM TCP/IP for MVS: Diagnosis Guide, LY43-0105*
- *IBM TCP/IP for MVS: Messages and Codes, SC31-7132*
- *IBM TCP/IP for MVS: Planning and Migration Guide, SC31-7189*

VS COBOL II

- *VS COBOL II Application Programming Guide for MVS and CMS, SC26-4045*
- *VS COBOL II Application Programming: Language Reference, GC26-4047*
- *VS COBOL II Installation and Customization for MVS, SC26-4048*

VS Fortran

- *VS Fortran Version 2: Language and Library Reference, SC26-4221*

- *VS Fortran Version 2: Programming Guide for CMS and MVS, SC26-4222*

VTAM®

- *Planning for NetView, NCP, and VTAM, SC31-8063*
- *VTAM for MVS/ESA Diagnosis, LY43-0069*
- *VTAM for MVS/ESA Messages and Codes, SC31-6546*
- *VTAM for MVS/ESA Network Implementation Guide, SC31-6548*
- *VTAM for MVS/ESA Operation, SC31-6549*
- *VTAM for MVS/ESA Programming, SC31-6550*
- *VTAM for MVS/ESA Programming for LU 6.2, SC31-6551*
- *VTAM for MVS/ESA Resource Definition Reference, SC31-6552*

Index

Special characters

– 398
% 398

A

abends 466
allocate functions
 AllocConnect 73
 AllocEnv 77
 AllocStmt 83
 NEW 152
 SQLAllocHandle 79
application
 compile 50
 execute 52
 execution steps 50
 link-edit 51
 multithreaded 421
 pre-link 51
 preparation 48
 requirements 49
 sample, DSNTJ8 50
 tasks 15
 trace 457
application variables
 binding 23
APPLTRACE keyword
 description 55
 use of 459
APPLTRACEFILENAME keyword 55
array input 403
array output 406
ASCII scalar function 471
ATRBEG service 396
ATREND service 396
attributes
 connection 389
 environment 389
 querying and setting 389
 statement 389
AUTOCOMMIT keyword 56

B

BINARY
 conversion to C 493
BindCol, function 85
binding
 application variables
 columns 24
 parameter markers 23
 packages 44
 plan 45
 sample, DSNTIJCL 43, 45
 stored procedures 45
BindParameter, function 91

BITDATA keyword 56

C

caching
 dynamic SQL statement 454
Cancel, function 102
case sensitivity 37
catalog
 functions
 description 397
 limiting use of 454
 querying 397
CHAR
 conversion to C 490
 display size 488
 length 487
 precision 485
 scale 486
character strings 35, 37
CLISHEMA keyword 56
 use of 400
ColAttribute, function 106
ColAttributes, function 114
COLLECTIONID keyword 57
column-wise binding 407
ColumnPrivileges, function 120
Columns, function 124
commit 26
common server 8
compile, application 50
CONCAT scalar function 471
configuring DB2 ODBC 41
CONNECT
 See also SQLDriverConnect
 type 1, type 2 18
Connect, function 129
connection attributes (options)
 description 389
 SetConnectAttr 336
 setting 452
connection attributes (options)
 GetConnectAttr 199
connection handle 9
 allocating 17
 AllocConnect, function 73
 freeing 17
 SQLAllocHandle, function 79
Connection handle
 Free, function 189
connection string 390
connectivity
 ODBC model 17
 requirements 42
CONNECTTYPE keyword 57
contexts
 multiple 424
coordinated distributed transactions 391

- coordinated transactions, establishing 394
- core level functions 7
- CURDATE scalar function 472
- CURRENTSERVER, plan bind option 45
- CURRENTSQLID keyword 58
- cursor
 - definition 26
 - function, SQLCloseCursor 104
 - use in DB2 ODBC 9
- CURSORHOLD keyword 58
- CURTIME scalar function 472

D

- data conversion
 - C data types 30
 - C to SQL data types 495
 - data types 30
 - default data types 30
 - description 34
 - display size of SQL data types 488
 - length of SQL data types 487
 - scale of SQL data types 486
 - SQL data types 30
 - SQL to C data types 489
- Data Conversion
 - precision of SQL data types 485
- data source information, querying 37
- data types
 - C 30, 33
 - generic 33
 - ODBC 33
 - SQL 30
- data-at-execute 401
- DATABASE scalar function 473
- DataSources, function 134
- DATE
 - conversion to C 493
 - display size 488
 - length 487
 - precision 485
 - scale 486
- DAYOFMONTH scalar function 472
- DB2 DataPropagator, managing shadow catalog 401
- DB2 ODBC
 - advantages of using 11
 - application requirements 48
 - components 48
 - configuring 41
 - diagnostic trace 459
 - initialization file 52
 - installing 41
 - migration considerations 65
 - shadow catalog 399
 - stored procedures 417
- DBNAME keyword 58
- DBRMs
 - binding to packages 43
- debugging 466
- DECIMAL
 - conversion to C 492

- DECIMAL (*continued*)
 - display size 488
 - length 487
 - precision 485
 - scale 486
- deferred arguments 23
- deprecated function 501
- DescribeCol, function 137
- diagnosis
 - description 28
 - function, GetDiagRec 223
 - trace 459
- DIAGTRACE keyword
 - description 59
 - use of 459
- DIAGTRACE_BUFFER_SIZE keyword 59
- DIAGTRACE_NO_WRAP keyword 59
- Disconnect, function 144
- DISCONNECT, plan bind option 45
- display size of SQL data types 488
- distributed transactions 391
- DOUBLE
 - conversion to C 492
 - display size 488
 - length 487
 - precision 485
 - scale 486
- driver
 - DB2 ODBC 467
 - ODBC 467
- driver manager 467
- DriverConnect, function 146
- DSN8O3VP, JCL sample 50, 508
- DSN8OIVP, JCL sample 50
- DSNAOINI
 - DD card 53
- DSNTEJ8, application sample 50
- DSNTIJCL, bind sample 43, 45
- DYNAMICRULES, package bind option 44

E

- embedded SQL
 - comparison to DB2 ODBC 9
 - mixing with DB2 ODBC 446
- environment
 - attributes (options) 389
 - information, querying 37
 - OS/390 UNIX, setup 46
 - runtime 41
- environment handle
 - allocating 16
 - AllocEnv, function 77
 - description 9
 - Free, function 191
 - freeing 16
 - NEW, function 152
- environmental variables
 - OS/390 UNIX 53
- error code, internal 466
- Error, function 155

- escape clauses, vendor 448
- example
 - stored procedure 508
- examples
 - array INSERT 405
 - catalog functions 399
- ExecDirect, function 161
- execute direct 22
- execute statement 22
- execute, application 50, 52
- Execute, function 166
- export statements
 - OS/390 UNIX 53
- ExtendedFetch, function 169

F

- FAR pointers 67
- Fetch, function 176
- fetching data in pieces 402
- FLOAT
 - conversion to C 492
 - display size 488
 - length 487
 - precision 485
 - scale 486
- ForeignKeys, function 181
- free functions
 - FreeConnect 189
 - FreeEnv 191
 - FreeStmt 196
- function list, ODBC 68
- function, deprecated 501
- functions
 - ODBC list 68, 69, 70, 71, 72

G

- GetConnectOption, function 202
- GetCursorName, function 204
- GetData, function 210
- GetFunctions, function 228
- GetInfo, function 234
- GetNumResultCols, function 294
- GetSQLCA, function 263
- GetStmtOption, function 273
- GetTypeInfo, function 278
- global transactions 56, 60, 396
- GRAPHIC
 - conversion to C 491
 - keyword 60

H

- handles
 - connection handle 9, 16
 - environment handle 9, 16
 - function, SQLFreeHandle 193
 - statement handle 9
- HOURL scalar function 472

I

- IFNULL scalar function 473
- initialization
 - file
 - common errors 55, 452
 - defaults, changing 390
 - description 52
 - specifying 53
 - task 15
- INSERT scalar function 471
- installation 41
- INTEGER
 - conversion to C 492
 - display size 488
 - length 487
 - precision 485
 - scale 486
- internal error code 466
- introduction to DB2 ODBC 7
- INVALID_HANDLE 28
- isolation levels, ODBC 470
- ISOLATION, package bind option 44

K

- keywords, initialization 54, 55

L

- LE threads
 - multiple 421
- LEFT scalar function 472
- length of SQL data types 487
- LENGTH scalar function 472
- link-edit, application 51
- long data
 - retrieving in pieces 401
 - sending in pieces 401
- LONGVARBINARY
 - conversion to C 493
- LONGVARCHAR
 - conversion to C 490
 - display size 488
 - length 487
 - precision 485
 - scale 486
- LONGVARGRAPHIC
 - conversion to C 491

M

- MAXCONN keyword 60
- metadata characters 398
- migration considerations 65
- MINUTE scalar function 472
- MONTH scalar function 472
- MoreResults, function 286
- MULTICONTEXT keyword 60
- multiple contexts 424
- multithreaded applications 421

MVSATTACHTYPE keyword 61
MVSDEFAULTSSID keyword 61

N

native error code 29
NativeSQL, function 290
notices, legal 535
NOW scalar function 472
null connect 419
null-terminated strings 35
NUMERIC
 conversion to C 492
 display size 488
 length 487
 precision 485
 scale 486
NumParams, function 292
NumResultCols, function 294

O

ODBC
 and DB2 ODBC 7, 467
 connectivity 17
 core level functions 7
 function list 68, 69, 70, 71, 72
 isolation levels 470
 vendor escape clauses 449
OPTIMIZEFORNROWS keyword 62
options
 connection 389
 environment 389
 querying and setting 389
 statement 389
OS/390 UNIX
 compile application 51
 environment setup 46
 environmental variable 53
 execute application 52
 export statements 53
 pre-link, link-edit application 51
 special considerations 50

P

packages, binding 43
ParamData, function 296
parameter markers
 array input 403
 binding 23
 use of 9
ParamOptions, function 298
PATCH2 keyword 62
pattern-values 398
plan, binding 45
PLANNAME keyword 62
pointers, FAR 67
portability 11
pre-link, application 51
precision of SQL data types 485

Prepare, function 300
prepare, statement 22
PrimaryKey, function 308
ProcedureCols, function 313
Procedures, function 323
PutData, function 327

Q

query statements, processing 24
querying
 data source information 37
 environment information 37
 shadow catalog 399
 system catalog information 397

R

REAL
 conversion to C 492
 display size 488
 length 487
 precision 485
 scale 486
registering stored procedures 418
remote site
 binding packages 45
REPEAT scalar function 472
result sets
 function generated 454
 retrieving into array 406
 returning from stored procedures 420
retrieving multiple rows 406
return codes 28
RIGHT scalar function 472
rollback 26
row-wise binding 407, 408
RowCount, function 330
ROWID
 conversion to C 495
 display size 488
 length 487
 precision 485
 scale 486
rowset 169
runtime environment
 setting up 43
 support 41

S

samples
 DSN8O3VP 50, 508
 DSN8OIVP 50
 DSNTEJ8 50
 DSNTIJCL 43, 45
scalar functions 451
scale of SQL data types 486
SCHEMALIST keyword 62
search arguments 398
SECOND scalar function 472

SELECT 24
 SetColAttributes, function 332
 SetConnectOption, function 345
 SetCursorName, function 347
 SetParam, function 354
 SetStmtOption, function 367
 shadow catalog
 managing 401
 querying 399
 SMALLINT
 conversion to C 492
 display size 488
 length 487
 precision 485
 scale 486
 SMP/E jobs 41
 SpecialColumns, function 369
 SQL
 dynamically prepared 10
 parameter markers 23
 preparing and executing statements 22
 query statements 24
 SELECT 24
 statements
 DELETE 26
 UPDATE 26
 VALUES 24
 SQL Access Group 7
 SQL_ATTR_ACCESS_MODE 337
 SQL_ATTR_AUTOCOMMIT 337
 SQL_ATTR_BIND_TYPE 361
 SQL_ATTR_CLOSE_BEHAVIOR 361
 SQL_ATTR_CONCURRENCY 361
 SQL_ATTR_CONNECTTYPE 337, 350
 SQL_ATTR_CURRENT_SCHEMA 337
 SQL_ATTR_CURSOR_HOLD 361
 SQL_ATTR_CURSOR_TYPE 361
 SQL_ATTR_MAX_LENGTH 361
 SQL_ATTR_MAX_ROWS 361
 SQL_ATTR_MAXCONN 337, 350
 SQL_ATTR_NODESCRIBE 361
 SQL_ATTR_NOSCAN 361
 SQL_ATTR_ODBC_VERSION 350
 SQL_ATTR_OUTPUT_NTS 350
 SQL_ATTR_PARAMOPT_ATOMIC 337
 SQL_ATTR_RETRIEVE_DATA 361
 SQL_ATTR_ROW_ARRAY_SIZE 361
 SQL_ATTR_ROWSET_SIZE 361
 SQL_ATTR_STMTTXN_ISOLATION 361
 SQL_ATTR_SYNC_POINT 337
 SQL_ATTR_TXN_ISOLATION 337, 361
 SQL_AUTOCOMMIT 452
 SQL_C_BINARY
 conversion from SQL 498
 SQL_C_BIT
 conversion from SQL 497
 SQL_C_CHAR
 conversion from SQL 496
 SQL_C_DBCHAR
 conversion from SQL 498
 SQL_C_DOUBLE
 conversion from SQL 497
 SQL_C_FLOAT
 conversion from SQL 497
 SQL_C_LONG
 conversion from SQL 497
 SQL_C_SHORT
 conversion from SQL 497
 SQL_C_TINYINT
 conversion from SQL 497
 SQL_C_TYPE_DATE
 conversion from SQL 498
 SQL_C_TYPE_TIME
 conversion from SQL 499
 SQL_C_TYPE_TIMESTAMP
 conversion from SQL 499
 SQL_COLUMN_AUTO_INCREMENT 115
 SQL_COLUMN_CASE_SENSITIVE 115
 SQL_COLUMN_CATALOG_NAME 115
 SQL_COLUMN_COUNT 115
 SQL_COLUMN_DISPLAY_SIZE 115
 SQL_COLUMN_DISTINCT_TYPE 115
 SQL_COLUMN_LABEL 115
 SQL_COLUMN_LENGTH 116
 SQL_COLUMN_MONEY 116
 SQL_COLUMN_NAME 116
 SQL_COLUMN_NULLABLE 116
 SQL_COLUMN_OWNER_NAME 116
 SQL_COLUMN_PRECISION 116
 SQL_COLUMN_QUALIFIER_NAME 115
 SQL_COLUMN_SCALE 116
 SQL_COLUMN_SCHEMA_NAME 116
 SQL_COLUMN_SEARCHABLE 117
 SQL_COLUMN_TABLE_NAME 117
 SQL_COLUMN_TYPE 117
 SQL_COLUMN_TYPE_NAME 117
 SQL_COLUMN_UNSIGNED 117
 SQL_COLUMN_UPDATABLE 117
 SQL_CONCURRENT_TRANS 392
 SQL_CONNECTTYPE 392
 SQL_COORDINATED_TRANS 392
 SQL_CURSOR_HOLD 453
 SQL_DATA_AT_EXEC 401
 SQL_ERROR 28
 SQL_MAX_ROWS 452
 SQL_NEED_DATA 28
 SQL_NO_DATA_FOUND 28
 SQL_NOSCAN 455
 SQL_NTS 35
 SQL_ROWSET_SIZE 407
 SQL_STMTTXN_ISOLATION 453
 SQL_SUCCESS 28
 SQL_SUCCESS_WITH_INFO 28
 SQL_TXN_ISOLATION 452
 SQLAllocConnect, function
 description 73
 SQLAllocEnv, function
 description 77
 SQLAllocHandle, function
 description 79

SQLAllocStmt, function
 description 83
 overview 20

SQLBindCol, function
 description 85
 overview 20, 24

SQLBindParameter, function
 description 91
 overview 24

SQLCancel, function
 description 102
 use in data-at-execute 402

SQLCloseCursor, function
 description 104

SQLColAttribute, function
 description 106

SQLColAttributes, function
 description 114
 overview 20, 24

SQLColumnPrivileges, function
 description 120

SQLColumns, function
 description 124

SQLConnect, function
 description 129

SQLDataSources, function
 description 134
 overview 20

SQLDescribeCol, function
 description 137
 overview 20, 24

SQLDescribeParam, function
 description 142

SQLDisconnect, function
 description 144

SQLDriverConnect, function
 description 146

SQLDriverConnect() 390

SQLEndTran, function
 description 152

SQLError, function
 description 155

SQLERROR, package bind option 44

SQLExecDirect, function
 description 161
 overview 20, 22

SQLExecute, function
 description 166
 overview 20, 22

SQLExtendedFetch, function
 description 169

SQLFetch, function
 description 176
 overview 20, 24

SQLForeignKeys, function
 description 181

SQLFreeConnect, function
 description 189

SQLFreeEnv, function
 description 191

SQLFreeHandle, function
 description 193

SQLFreeStmt, function
 description 196
 overview 20

SQLGetConnectAttr, function
 description 199

SQLGetConnectOption, function
 description 202

SQLGetCursorName, function
 description 204

SQLGetData, function
 description 210
 overview 20, 24

SQLGetDiagRec, function
 description 223

SQLGetEnvAttr, function
 description 226

SQLGetFunctions, function
 description 228

SQLGetInfo, function
 description 234

SQLGetLength, function
 description 257

SQLGetPosition, function
 description 259

SQLGetSQLCA, function
 description 263

SQLGetStmtAttr, function
 description 270

SQLGetStmtOption, function
 description 273

SQLGetSubString, function
 description 275

SQLGetTypeInfo, function
 description 278

SQLMoreResults, function
 description 286
 use of 404

SQLNativeSql, function
 description 290

SQLNumParams, function
 description 292

SQLNumResultCols, function
 description 294
 overview 20, 24

SQLParamData, function
 description 296
 use in data-at-execute 401

SQLParamOptions, function
 description 298

SQLPrepare, function
 description 300
 overview 20, 22

SQLPrimaryKeys, function
 description 308

SQLProcedureColumns, function
 description 313

SQLProcedures, function
 description 323

- SQLPutData, function
 - description 327
 - use in data-at-execute 401
- SQLRowCount, function
 - description 330
 - overview 20
- SQLSetColAttributes, function
 - description 332
- SQLSetConnectAttr, function
 - description 336
- SQLSetConnection, function
 - description 343
- SQLSetConnectOption, function
 - description 345
- SQLSetCursorName, function
 - description 347
- SQLSetEnvAttr, function
 - description 350
- SQLSetParam, function
 - description 354
 - overview 20, 22, 24
- SQLSetStmtAttr, function
 - description 360
- SQLSetStmtOption, function
 - description 367
- SQLSpecialColumns, function
 - description 369
- SQLSTATE
 - description 29
 - format of 29
 - function cross reference 475
 - in DB2 ODBC 9
- SQLStatistics, function
 - description 374
- SQLTablePrivileges, function
 - description 379
- SQLTables, function
 - description 382
- SQLTransact, function
 - description 386
 - overview 20, 24, 26
- SRRBACK service 396
- SRRCMIT service 396
- statement attributes (options)
 - description 389
 - function, SetStmtAttr 360
 - GetStmtAttr 270
 - setting 452
- statement handle 9
 - allocating 22
 - Free, function 196
 - freeing 28
 - maximum number of 22
 - SQLAllocHandle, function 79
- statement handle functions
 - AllocStmt 83
- Statistics, function 374
- stored procedures
 - binding 45
 - catalog table
 - registering 418

- stored procedures (*continued*)
 - example 508
 - ODBC escape clause 451
 - returning result sets 420
 - tracing 462
 - using with DB2 ODBC 417
- string
 - arguments 35, 37
 - null-termination 35
 - truncation 36
- SUBSTRING scalar function 472
- subsystem, defining 52
- suffix-W APIs
 - description 429
 - setup 46
- SYSSHEMA keyword 63
- system catalogs, querying 397

T

- Table, function 382
- TablePrivileges, function 379
- TABLETYPE keyword 64
- termination task 15
- threads
 - multiple 421
- THREADSAFE keyword
 - description 64
- TIME
 - conversion to C 493
 - display size 488
 - length 487
 - precision 485
 - scale 486
- TIMESTAMP
 - conversion to C 494
 - display size 488
 - length 487
 - precision 485
 - scale 486
- trace
 - application 457
 - DB2 ODBC diagnosis 459
 - keywords, using 459
 - stored procedure 462
- Transact, function 386
- transaction
 - isolation levels, ODBC 470
 - management 26
 - processing 15
- truncation 36
- TXNISOLATION keyword 64

U

- UNDERSCORE keyword 65
- Unicode
 - application variables 23
 - UCS-2 setup 46
 - UCS-2 support 429
- USER scalar function 473

V

VALUES 24

VARBINARY

conversion to C 493

VARCHAR

conversion to C 490

display size 488

length 487

precision 485

scale 486

VARGRAPHIC

conversion to C 491

vendor escape clauses 448

W

writing DB2 ODBC applications 15

X

X/Open CAE 29

X/Open Company 7

X/Open SQL CLI 7

Readers' Comments — We'd Like to Hear from You

DB2 Universal Database for OS/390 and z/OS
ODBC Guide and Reference
Version 7

Publication No. SC26-9941-03

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>				

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>				
Complete	<input type="checkbox"/>				
Easy to find	<input type="checkbox"/>				
Easy to understand	<input type="checkbox"/>				
Well organized	<input type="checkbox"/>				
Applicable to your tasks	<input type="checkbox"/>				

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? Yes No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.



Fold and Tape

Please do not staple

Fold and Tape



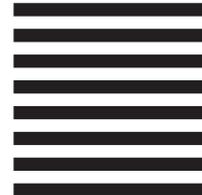
NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
H150/090
555 Bailey Avenue
San Jose, CA 95141-9989
U. S. A.



Fold and Tape

Please do not staple

Fold and Tape



Program Number: 5675-DB2

Printed in U.S.A.

SC26-9941-03

