DB2 DataJoiner

Generic Access API Reference

Version 2 Release 1

SC26-9147-00

DB2 DataJoiner

Generic Access API Reference

Version 2 Release 1

SC26-9147-00

First Edition (October 1997)

This edition applies to Version 2 of IBM DB2 DataJoiner, 5765-C36, 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.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, W92/H3
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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 of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (1) the exchange of information between independently created programs and other programs (including this one) and (2) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department J01, 555 Bailey Avenue, San Jose, CA 95161-9023. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Advanced Peer-to-Peer Networking (APPN)
Advanced Program-to-Program Communication (APPC)
AIX
AIXwindow
AIX/6000
APPN
AS/400
Customer Information Control System (CICS)
CICS/6000
DATABASE 2
DataGuide
DataJoiner
DataPropagator
DB2
DB2/2
DB2/400
DB2/6000
DFSMS
Distributed Relational Database Architecture
DRDA
Extended Services for OS/2
IBM
IIN
IMS
MVS
IMS/ESA
MVS/ESA
MVS/XA
Operating System/2
Operating System/400
OS/2
OS/390
OS/400
RACF
RS/6000
SQL/DS
SQL/400
System/390
System Modification Program Extended (SMP/E)
VisualAge
VisualGen
Virtual Telecommunications Access Method (VTAM)

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

Intel and Pentium are trademarks of the Intel Corporation.

HP-UX is a registered trademark of Hewlett-Packard.

Solaris is a registered trademark of Sun Microsystems, Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Cross-Platform Terminology Conventions

Some product names in the documentation refer to more than one product; other names refer to specific product levels. This list contains the most frequently used product definitions.

DataJoiner Refers to DB2 DataJoiner Version 2. References specific to or including DataJoiner Version 1 will include the version.

DB2 Family Refers to all DataJoiner- supported versions of DATABASE 2 (DB2) database server products on all platforms (DB2 for OS/390, DB2 for VM, DB2 for common servers, DataJoiner, and so on). Supported versions are listed in the *DataJoiner Planning, Installation, and Configuration Guide* for your platform.

DB2 By itself, refers to any one or all of the DB2 for common server Version 2 database server products on all platforms, which includes DataJoiner.

If a DB2 reference is qualified with a specific operating system or version, the reference applies only to that particular version.

DB2 for CS Refers to any DB2 for common servers Version 2 database server product. This term is often used when describing DataJoiner and DB2 for common servers functional differences

About This Library

To understand the organization of the DataJoiner library, it is important to understand the relationship between DataJoiner and DB2 for CS. DataJoiner provides a “superset” of DB2 for CS. The two products share common functions and syntax; therefore, information that is common to DataJoiner and DB2 for CS is documented in the DB2 for CS books. The DataJoiner books listed in Table 1 document the function and syntax that DataJoiner has *in addition to* the function and syntax it shares with DB2 for CS.

DataJoiner, DB2 for CS, and Replication Library Publications

Table 1 lists the DataJoiner, DB2 for CS, and Replication manuals applicable to installing, configuring, administrating, using, and running applications against DataJoiner. The *DataJoiner for AIX Planning, Installation, and Configuration Guide* and the *DataJoiner for Windows NT Systems Planning, Installation, and Configuration Guide* books are provided with DataJoiner. The remaining books are provided in softcopy formats on the product CD-ROM. All listed books are provided in PostScript; most are provided in HTML (the two exceptions are the DB2 for CS Software Developer Kit publications). Additionally, most of the DB2 for CS books are provided in INF format (see Table 1).

Table 1 does not list all of the DB2 for CS books. View or print a DB2 for CS book to see the publications list for all DB2 for CS books.

If you order Classic Connect, you will receive additional books (the *DataJoiner Classic Connect Planning, Installation, and Configuration Guide*, the *DataJoiner Classic Connect data mapper Sample for Windows Installing and Using Guide*, and the *DataJoiner Messages and Problem Determination Guide*) and a program directory.

Table 1 (Page 1 of 4). DataJoiner, DB2 for CS, and Replication publications applicable to DataJoiner

Book Name	Form Number	File Prefix	INF
DataJoiner Version 2.1 Books			
DataJoiner for Windows NT Systems Planning, Installation, and Configuration Guide	SC26-9150	DJXN1	no
This book covers capacity planning, resource management, installation, and configuration tasks for IBM DataJoiner on Microsoft Windows NT operating systems.			
DataJoiner for AIX Systems Planning, Installation, and Configuration Guide	SC26–9145	DJXG5	no
This book covers capacity planning, resource management, installation, and configuration tasks for IBM DataJoiner on AIX operating systems.			
DataJoiner Administration Guide	SC26–9146	DJXD4	no

Table 1 (Page 2 of 4). DataJoiner, DB2 for CS, and Replication publications applicable to DataJoiner

Book Name	Form Number	File Prefix	INF
This book provides information that assists DBAs and other system administrators of DataJoiner, IBM's heterogeneous data access product, to perform administrative tasks. It includes a product overview section, security considerations, data source identification steps, database utility notes, performance considerations, database system monitor reference data, large object information and explain tool examples.			
DataJoiner Application Programming and SQL Reference Supplement	SC26–9148	DJXK4	no
This book provides SQL statements, descriptions of system catalog data, guidelines, and other information for application programmers. With this information, application programmers can use DataJoiner to perform multiple tasks in a distributed database environment—tasks such as creating nicknames by which to reference tables and views, invoking functions and stored procedures, passing SQL directly to databases for processing, and using server options to optimize query performance.			
DataJoiner Generic Access API Reference	SC26–9147	DJXM4	no
This book explains how to create a generic access module that allows you to use existing drivers or to create new drivers to gain access to an unlimited set of data sources.			
DataJoiner Classic Connect Planning, Installation, and Configuration Guide	GC26–8869	DJXC4	no
This book provides information on the DataJoiner Classic Connect for MVS product. The audience for this information includes application programmers, database administrators, network administrators, system administrators, and system programmers. The book documents key tasks required to set up Classic Connect in the MVS operating environment: planning your setup, installing components via SMP/E, configuring the kernel, DMSIs, and network communications, managing instances, and creating relational data maps for IMS and VSAM data.			
DataJoiner Classic Connect data mapper Sample for Windows Installing and Using Guide	GC26–8873	DJXZ2	no
This book provides information on the DataJoiner Classic Connect data mapper sample for Windows. The audience for this information includes system programmers, DBAs, or anyone that needs to produce relational maps (USE grammar) for IMS and VSAM data. The book documents key tasks required to set up and use the data mapper in the Windows environment: installing product files, starting the product, and generating USE grammar statements for input to DataJoiner Classic Connect projection utilities.			
DataJoiner Messages and Problem Determination Guide	SC26–9149	DJXP4	no
This book describes the messages and codes issued by DataJoiner and Classic Connect instances. For messages that report errors, the book explains the cause of the errors and recommends corrective actions. The book also provides guidelines on using diagnostic tools to isolate and understand problems.			
DB2 for CS and Replication Books			
DB2 Information and Concepts Guide	SH20–4664	SQLG0	no

Table 1 (Page 3 of 4). DataJoiner, DB2 for CS, and Replication publications applicable to DataJoiner

Book Name	Form Number	File Prefix	INF
Provides product and conceptual information to anyone who needs a comprehensive overview of the DB2 products. It is useful when deciding which DB2 products suit your environment. It also includes a glossary of terms used in the book			
DB2 Administration Guide	S20H-4580	SQLD0	yes
Contains information required to design, implement, and maintain a database to be accessed either locally or in a client/server environment.			
DB2 Database System Monitor Guide and Reference	S20H–4871	SQLF0	yes
Includes a description of how to use the Database System Monitor and a description of all the data elements for which information can be collected.			
DB2 Command Reference	S20H–4645	SQLN0	yes
Provides the reference information needed to use system commands and the DB2 command line processor to execute database administrative functions. Describes the commands that can be entered at an operating system command prompt or in a shell script to access the database manager. Explains how to invoke and use the command line processor, and describes the command line processor options. Provides a description of all the database manager commands.			
DB2 API Reference	S20H–4984	SQLB0	yes
Provides information about the use of application programming interfaces (APIs) to execute database administrative functions. Presents a description of APIs and the data structures used when calling APIs, as well as detailed information on the use of database manager API calls in applications written in the supported programming languages.			
DB2 SQL Reference	S20H–4665	SQLS0	yes
Is intended to serve as a reference for syntax and rules governing the use of SQL statements. Syntax diagrams, semantic descriptions, rules and examples are provided for the SQL statements. Catalog views, product maximums, release-to-release incompatibilities, and a glossary are also included in this book.			
DB2 Application Programming Guide	S20H–4643	SQLA0	yes
Discusses the application development process and how to code, compile, and execute application programs that use embedded SQL to access the database. It includes discussions on programming techniques and performance considerations for the application programmer.			
DB2 Call Level Interface Guide and Reference	S20H–4644	SQLL0	yes
Is a guide and reference manual for programmers using the Call Level Interface. DB2 Call Level Interface is a callable SQL interface based on the X/Open** CLI specification and is compatible with Microsoft** Corporation's ODBC.			
DB2 Messages Reference	S20H–4808	SQLM0	yes
Lists messages and explanations. Each explanation includes the action to be taken when a message or code is issued.			

Table 1 (Page 4 of 4). DataJoiner, DB2 for CS, and Replication publications applicable to DataJoiner

Book Name	Form Number	File Prefix	INF
DB2 Problem Determination Guide	S20H–4779	SQLP0	yes
Provides information that helps in determining the source of errors, recovering from problems, and describing and reporting defects.			
DDCS User's Guide	S20H–4793	SQLC0	yes
Provides concepts, programming and general information about the DDCS products.			
DB2 Replication Guide and Reference	S95H–0999	DB2E0	no
Describes how to plan, configure, administer, and operate IBM replication products, including Apply and Capture.			
DB2 for CS Platform-Specific Books			
DB2 SDK for AIX Building Your Applications	S20H-4780	SQLA3	yes
This manual provides environment setup information and step-by-step instructions to compile and link DB2 applications on an Windows 95 and NT, Version 2.			
DB2 SDK for Windows 95 and NT Building Your Applications	S33H-0310	SQLA6	yes
This manual provides environment setup information and step-by-step instructions to compile and link DB2 applications on an Windows 95 and NT, Version 2.			

Publication Ordering, Viewing, and Printing Instructions

Use order number SBOF-5289 to request one hardcopy of each of the DataJoiner, DB2 for CS, and Replication books shown in the previous sections.

To view online documentation, follow the instructions located in the README files on the CD-ROM. Most of the books (in Table 1 on page ix) are provided as HTML files and can be viewed with an HTML browser. You can also view INF versions of many DB2 for CS books. Instructions for installing the INF reader on AIX is provided in the DB2 README files; on NT operating systems, the INF reader is installed automatically. DataJoiner and Replication information is not provided in INF format.

To print individual books, follow the instructions provided in the README files on the CD-ROM. PostScript files for all the books shown in previous sections are provided.

World Wide Web and Internet Information Resources

The following electronic resources provide additional information about DataJoiner.

World Wide Web The following DataJoiner-specific web site contains general and technical (frequently-asked-questions) product information. The HTTP address is:

`http://www.software.ibm.com/data/datajoiner/`

Internet Newsgroups DataJoiner questions, answers, and discussions can be found in:

- bit.listserv.db2-l
- comp.databases
- comp.databases.ibm-db2

What's New in DataJoiner Version 2?

DataJoiner Version 2 offers many new features and enhancements. This chapter describes some of the major changes for this version, and points you to sources of more information in the DataJoiner and DB2 libraries. Major enhancements include:

DB2 Version 2 functionality

DataJoiner is built on the DB2 Version 2 code base, which means that DataJoiner provides all the major functional enhancements provided by DB2, including:

- Extended SQL capabilities
- An enhanced SQL optimizer
- Improved database performance
- Systems management support
- Robust integrity and data protection
- Object relational capabilities
- National language support (NLS)

See the *DB2 Administration Guide* for detailed information about these features.

DataJoiner for Windows NT

DataJoiner has extended its reach to provide an industrial strength heterogeneous database management system to the Windows NT platform. DataJoiner for Windows NT supports the same SQL and features as DataJoiner for UNIX-based platforms.

Expanded DataJoiner SQL support

This version of DataJoiner contains many new and modified SQL statements. New DDL statements provide greater flexibility and safety in defining your DataJoiner environment—users can create, alter, and drop mappings for data sources, users, user-defined and built-in functions, and data types. Additionally, new SQL DML statements provide enhanced functions for local and distributed queries; an example is the CASE expression, which is useful for selecting an expression based on the evaluation of one or more conditions.

Distributed heterogeneous update support

DataJoiner now allows you to update multiple heterogeneous data sources within a distributed unit of work while maintaining transaction atomicity. This task is accomplished through adherence to the two-phase commit model. Supported data sources include most versions of the DB2 Family and, with the appropriate XA libraries, various other data sources as well.

New graphical installation, configuration, and administration tools

A variety of new tools is available to help you accomplish most administrative chores. TaskGuides walk you through common tasks, such as configuring communications and data source access. The Administrator's Toolkit provides a collection of tools designed to assist you with the day-to-day operation of DataJoiner. It consists of the following components:

The Command Line Processor A system command prompt used to access and manipulate databases.

The Database Director Allows you to perform configuration, backup and recovery, directory management, and media management tasks.

Visual Explain A tool for graphically viewing and navigating complex SQL access plans.

The DB2 Performance Monitor Monitors the performance of your DB2 system for tuning purposes.

Stored procedures

DataJoiner now supports stored procedures at remote data sources as well as the local DataJoiner database. Use stored procedures to speed application performance. For example, applications that process huge amounts of data at a server but return smaller result sets should run faster as stored procedures. Another benefit is that stored procedures usually reduce network traffic between clients and databases.

DataJoiner stored procedures can augment standard data security. For example, in a 3-tier environment, data can be retrieved from a remote server and then processed at the DataJoiner server; only a subset of data needs to available to the client.

Heterogeneous data replication

DataJoiner now provides replication administration as an integrated component. You can define, automate, and manage replication data from a single control point across your enterprise. A GUI provides administrative support for the replication environment, with objects and actions that define and manage source and target table definitions. DataJoiner's Apply component performs the actual replication, tailoring and enhancing data as you specify, and serving as the interface point to and from your various data sources.

System catalog information available in views

DataJoiner provides views from which you can access system catalog information about each DataJoiner database. Some of these views contain data—for example, data about tables, indexes, and servers—that was accessible only from tables in previous versions of DataJoiner. Other views contain data—for example, data about stored procedures, server options, and server functions—that is now available in Version 2.

RDB Support

With Version 2, DataJoiner continues to increase the number of natively supported data sources. Oracle RDB is the most recent addition.

Performance Enhancements

In addition to general engine performance improvements, this latest version offers new query rewrite capabilities, improved pushdown performance, and remote query caching.

Chapter 1. Introduction to the Generic Access API

IBM DataJoiner is a multidatabase server that provides client access to diverse data sources that reside on different platforms. DataJoiner supports a variety of data sources; however there might be times when you want to access a source that DataJoiner doesn't support. To access these data sources you can create your own connection to a specific data source.

DataJoiner provides two methods of accessing data sources that it does not natively support.

- Use of a third-party gateway such as CrossAccess
- Creation of a custom data access module for that data source by either purchasing an Open Database Connectivity (ODBC) or X/Open CLI-compliant driver for your platform or writing a driver that supports the Generic Access API defined in this manual

This book describes how you can create a custom data access module with DataJoiner's generic access API.

Example of a Custom Data Access Module

To take a fictional example: Suppose that Datex is a type of data source that DataJoiner doesn't support. To enable DataJoiner to access data stored in a Datex data source, you must create a custom data access module.

A custom data access module consists of DataJoiner's generic access API component and a driver, as shown in Figure 1.

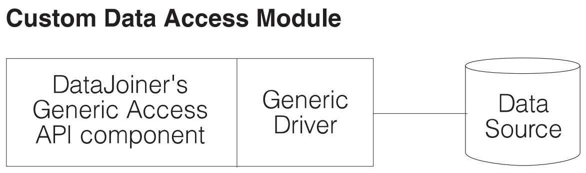
Custom Data Access Module
DataJoiner's Generic Access API component
Generic Driver
Data Source

Figure 1. Custom Data Access Module

To create a custom data access module, you must:

1. Supply a driver that supports the generic access API. This can be a driver you purchase that is X/Open CLI compliant or provides at least the ODBC core level of function. It can also be a generic driver you write that supports the generic access API defined in this manual. The driver is installed on the same system as DataJoiner, which means that any communication with the server containing the data is left to the driver you provide.

   The X/Open CLI and Microsoft's ODBC are specifications for call level interfaces that provide application programmers with an alternative to dynamic embedded SQL. An X/Open CLI or ODBC compliant driver supports the specifications so that programmers can build and run applications using the function calls defined by the specifications.

DataJoiner's Generic Access API is a specification for the function calls that the Generic Access API component uses to perform operations against a data source. This specification is a subset of the X/Open CLI and ODBC. Therefore, DataJoiner can use an X/Open CLI or ODBC compliant driver.

2. Link-edit the driver with DataJoiner's generic access API component.
3. Configure DataJoiner so it recognizes your new access module. Information about configuring DataJoiner is in the *DataJoiner for AIX Planning, Installation, and Configuration Guide*.
4. Test your access module by using DataJoiner's pass-through capability to exercise function. Later you can define and use nicknames.

In addition to reading and understanding this manual, familiarize yourself with the Microsoft ODBC interface. Before you get started on the steps described above, you need some information on the API, which you will get in the next section.

Elements of the Generic Access API

The name of each function supported by the generic access API starts with the prefix “SQL.” Each function includes one or more arguments.

Functions

The generic access API includes required and optional functions. DataJoiner requires the following functions:

SQLAllocConnect
SQLAllocEnv
SQLAllocStmt
SQLBindCol
SQLConnect
SQLDescribeCol
SQLDisconnect
SQLError
SQLExecDirect
SQLExecute
SQLFetch
SQLFreeConnect
SQLFreeEnv
SQLFreeStmt
SQLGetCursorName
SQLNumResultCols
SQLPrepare
SQLRowCount
SQLSetParam
SQLTransact

The optional functions are:

- SQLColumns
- SQLDriverConnect

Required functions are described in Chapter 3, “Required Functions” on page 13.

Buffers

DataJoiner passes data to a generic driver in an input buffer and retrieves data from an output buffer. DataJoiner allocates memory for both input and output buffers. When

DataJoiner uses the buffer to send or retrieve string data, the buffer includes space for the null termination byte.

Input Buffers

DataJoiner passes the address and length of an input buffer to a generic driver. The length of the buffer will be one of the following values:

- A length greater than or equal to zero. This is the actual length of the data in the input buffer (excluding the null termination byte if the data is a string). For string data, a length of zero indicates that the data is an empty (zero length) string. This is different from a null pointer.
- SQL_NTS. This specifies that the data value is a null-terminated string.
- SQL_NULL_DATA. This tells the generic driver to ignore the value in the input buffer and use a NULL data value instead. It is valid only when the input buffer is used to provide the value of a parameter in an SQL statement.

Unless it is specifically prohibited in the description of a given function, the address of an input buffer can be a null pointer. When the address of an input buffer is a null pointer, the value of the corresponding buffer length argument must be ignored.

Output Buffers

DataJoiner passes the following arguments to the generic driver, so that it can return data in an output buffer:

- The address of the buffer.
- The length of the buffer. This can be ignored by the generic driver if the returned data has a fixed width in C, such as an integer, real number, or date structure.
- The address of a variable in which the generic driver must return the length of the data. The returned length of the data is SQL_NULL_DATA if the data is a NULL value in a result set. Otherwise, it is the available number of bytes of data (not including the null termination byte if the data is a string).

If the output buffer is too small, the generic driver should attempt to truncate the data. It should truncate the data and return SQL_SUCCESS_WITH_INFO if the truncation does not cause a loss of significant data. It should leave the buffer untouched and returns SQL_ERROR if the truncation will cause a loss of significant data. DataJoiner will call SQLERROR to retrieve information about the truncation or the error.

Environment, Connection, and Statement Handles

The generic driver can allocate storage for information about the generic access API environment, each connection, and each SQL statement. The handles to these storage areas are returned to DataJoiner. DataJoiner uses one or more of these handles in each call to a function.

The generic access API defines three types of handles:

- An *environment handle* identifies memory storage for global information, including the valid connection handles and current active connection handle. It is a variable of type HENV. DataJoiner uses a single environment handle per use; it will request this handle prior to connecting to a data source.

- *Connection handles* identify memory storage for information about a particular connection. They are variables of type HDBC. DataJoiner will request a connection handle prior to connecting to a data source. Each connection handle is associated with the environment handle. Each environment handle can, however, have multiple statement handles associated with it.
- *Statement handles* identify memory storage for information about an SQL statement. They are variables of type HSTMT. DataJoiner will request a statement handle prior to submitting SQL requests. Each statement handle is associated with exactly one connection handle. Each connection handle can, however, have multiple statement handles associated with it.

For more information on connection handles, see “Establishing a Connection to a Data Source” on page 6. For more information on statement handles, see “Processing an SQL Statement” on page 6.

Data Type Support

The generic access API defines SQL data types and C data types. A generic driver should support these data types in the following ways:

- Accepts DataJoiner SQL and C data types as arguments in function calls.
- Translates DataJoiner SQL data types to SQL data types acceptable by the data source, if necessary.
- Converts C data from DataJoiner to the SQL data type required by the data source.
- Converts SQL data from a data source to the C data type requested by DataJoiner.
- Provides access to data type information through the SQLDescribeCol function.

For more information on data types, see Appendix B, “Data Conversion” on page 95. The C data types are defined in `sqlcli.h` found in Appendix C, “Command Line Interface Include File” on page 115.

Function Return Codes

The generic driver must return a predefined code after function execution. These return codes should indicate success, warning, or failure status. The return codes expected are:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_ERROR
- SQL_INVALID_HANDLE

If the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, DataJoiner can call SQLError to retrieve additional information. For a complete description of return codes and error handling, see “Returning Status and Error Information” on page 10.

Chapter 2. Writing a Generic Driver

Each generic driver must support the required generic access API functions. These functions perform tasks such as allocating and deallocating memory, transmitting or processing SQL statements, and returning results and errors.

Role of DataJoiner

DataJoiner calls the generic driver when needed and performs the loading and unloading of the custom data access module. DataJoiner uses only valid function arguments and controls state transitions.

Required function calls must be supported by the generic driver. If a function is not supported, a custom data access module cannot be created using that generic driver.

Arguments

The following items describe the valid arguments or types of arguments used by DataJoiner.

- Environment, connection, and statement handles are not null pointers and are the correct type of handle for the argument.
- Other required arguments are not null pointers.
- Option flags that cannot be extended by a generic driverspecify only valid options.
- Argument values that specify a column or parameter number are greater than 0. The generic driver should check the upper limit of these argument values based on the limits of the data source to which they provide access.
- Buffer length arguments values are appropriate for the corresponding buffer in the context of the given function.

State Transitions

The following items discuss the state transitions controlled by DataJoiner.

- The state of the *hdbc* is valid in the context of the function's requirement.
- Function calls are checked to ensure that the order in which functions are called is valid.
- Function calls comply with the state transitions found in Appendix D, “State Transition Tables” on page 124.

Establishing Connections

This section describes how DataJoiner and the generic driver work together to establish a connection to a data source.

Data Sources

A data source consists of the data a user wants to access, its associated DBMS, the platform on which the DBMS resides, and the network (if any) used to access that platform. The generic driver must provide certain information to the data source in order to connect to it. At the core level, this is defined to be the name of the data source, a user ID, and a password.

Establishing a Connection to a Data Source

All generic drivers must support the following connection-related functions:

- SQLAllocEnv allows the generic driver to allocate storage for environment information.
- SQLAllocConnect allows the generic driver to allocate storage for connection information.
- SQLConnect allows an application to establish a connection with the data source. DataJoiner passes the following information in the call to SQLConnect:
  - Data source name
  - User ID - optional
  - Authentication string (password) - optional

Table 2 shows the command flow between DataJoiner and a generic driver.

Table 2. Connection Flow Between DataJoinerand a Generic Driver

DataJoiner Executes	Generic Driver Executes
SQLAllocEnv	
SQLAllocConnect	
SQLSetConnectOption	
SQLConnect	SQLAllocEnv → SQLAllocConnect → SQLSetConnectOption → SQLConnect
processing SQL	processing SQL
SQLDisconnect	SQLDisconnect → SQLDisconnect → SQLFreeConnect → SQLFreeEnv
SQLFreeConnect	
SQLFreeEnv	

Processing an SQL Statement

Figure 2 on page 7 shows a simple sequence of calls to execute SQL statements.

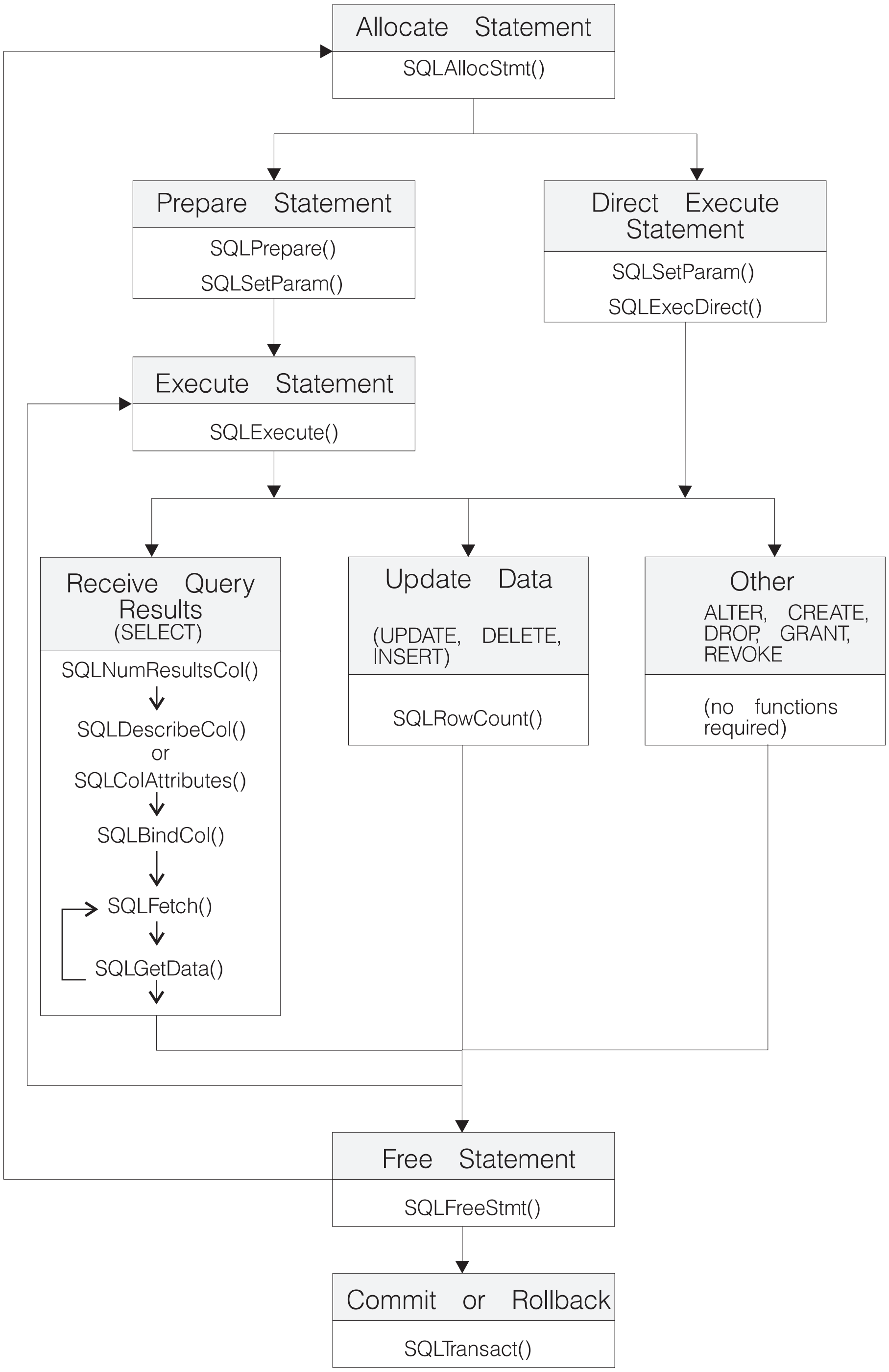

Figure 2. SQL Processing

Allocating a Statement Handle

Before DataJoiner can submit an SQL statement, it calls SQLAllocStmt to request that the generic driver allocate storage for the statement. DataJoiner passes a connection handle and the address of a variable of type HSTMT to the generic driver. The generic driver must allocate storage for the statement, associate a statement with the connection referenced by the connection handle, and return the statement handle in the variable.

A generic driver must use the statement handle to reference storage for names, parameter and binding information, error messages, and other information related to a statement processing stream.

Executing an SQL Statement

DataJoiner can submit an SQL statement for execution in two ways:

- Prepared — used if the same SQL statement will be executed more than once or if information about the result set is needed prior to execution.
- Direct — used if an SQL statement will be executed only once and no information about the result set is needed prior to execution.

Prepared Execution

If supported by the data source, preparing a statement before it is executed has the following advantages:

- It is the most efficient way to execute the statement more than once, especially if the statement is complex. The data source compiles the statement, produces an access plan, and returns an access plan identifier to the data access module. The data source minimizes processing time by using the access plan each time it executes the statement.
- It allows the generic driver to send an access plan identifier instead of an entire statement each time the statement is executed. This minimizes network traffic.
- The generic driver can return information about a result set before executing the statement.

If the data source does not support prepared execution, the generic drivermust emulate it to the extent possible.

A generic driver must support prepared execution through SQLPrepare, SQLSetParam, and SQLExecute. SQLPrepare and SQLSetParam can be called in any order after SQLAllocStmt has been called and before SQLExecute is called.

Direct Execution

The SQLExecDirect function supports direct execution. If necessary, the generic driver can translate the statement to the form of SQL used by the data source. It then sends the SQL string to the data source.

Supporting Transactions

Generic drivers should support manual-commit for transactions.

Extensions for SQL Statements

This section describes extension functions related to processing SQL statements.

To return information about data, a generic driver can support the catalog function, SQLColumns. SQLColumns returns a result set that lists the column names in a specified table. DataJoiner retrieves these results by calling SQLBindCol and SQLFetch.

DataJoiner will submit the following date, time and timestamp syntax to the generic driver:

- date *yyyy-mm-dd*
- time *hh12:mm:ss AM* or *hh24:mm:ss*
- timestamp *hh12:mm:ss.nnnnnn AM* or *hh24:mm.ss.nnnnnn*

The generic driver does not need to check the validity of the syntax except as needed to translate it to syntax specific to the data source.

Returning Results

A generic access API function, such as SQLColumns, returns data. Other SQL statements do not return result sets. For these statements, the code returned by the generic driver from SQLExecute or SQLExecDirect is usually the only source of information as to whether the statement was successful.

DataJoiner might or might not know the form of an SQL statement prior to execution. Therefore, generic drivers must support functions that allow DataJoiner to request information about the result set.

Binding

DataJoiner assigns storage for result columns before or after submitting an SQL statement. A generic driver must bind storage to result columns on the basis of information passed to it through SQLBindCol. The generic driver should:

- Accept pointer arguments that reference storage areas.
- Associate each column with the given storage area.
- Store information about the data type to which to convert the result data.

The generic driver must use this information during subsequent fetch operations.

Determining the Characteristics of a Result Set

Each generic driver must support the following core functions:

- SQLNumResultsCols returns the number of columns in the result set.
- SQLDescribeCol provides information about a column in the result set.

- SQLRowCount returns the number of rows affected by an SQL statement.

Returning Result Data

DataJoiner binds columns of the result set to storage locations with SQLBindCol. It retrieves a row of data with SQLFetch. Each time SQLFetch is called, a generic driver should:

1. Move the cursor to the next row.
2. Retrieve the data from the data source.
3. Convert the data for each bound column to the form specified by the *fCType* argument in SQLBindCol. The generic driver might need to truncate the data for some data type conversions.
4. Place the converted data for each bound column in the storage pointed to by the *rgbValue* argument in SQLBindCol. For some data types, the generic driver might truncate the data if the storage location is too small. See Appendix B, “Data Conversion” on page 95.

Supporting Cursors

A generic driver must support multiple simultaneous cursors per connection. A generic driver must maintain a cursor to keep track of its position in the result set. Each time DataJoiner calls SQLFetch, the generic driver should move the cursor to the next row and return that row. The cursor only scrolls forward, one row at a time.

Positioned update and delete statements require cursor names. The generic driver must generate the cursor name and associate it with the SQL statement. To retrieve the cursor name for an *hstmt*, DataJoiner calls SQLGetCursorName.

Returning Status and Error Information

This section presents information about return codes and error messages.

Return Codes

When DataJoiner calls a function, the generic driver must return a predefined code. These return codes indicate success, warning, or failure status. The following table lists all possible return codes for the generic access API functions.

Table 3. 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 `SQLError()` to receive the SQLSTATE and any other error information. The SQLSTATE will have a class of '01', see Table 47 on page 89.
SQL_NO_DATA_FOUND	The function returned successfully, but no relevant data was found.
SQL_ERROR	The function failed. Call `SQLError()` to receive the SQLSTATE and any other error information.
SQL_INVALID_HANDLE	The function failed due to an invalid input handle (environment, connection or statement handle).

Error Messages

If a function other than SQLError returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, DataJoiner will call SQLError to obtain additional information. Additional error or status information can come from one of two sources:

- Error or status information from a function, indicating that a programming error was detected.
- Error or status information from the data source, indicating that an error occurred during SQL statement processing.

The generic driver must buffer errors or messages for the function it is currently executing. The generic driver's error buffer should store multiple errors for a function. After the generic driver has executed the function, DataJoiner can call SQLError to return error messages for the function. Each time DataJoiner calls SQLError, the generic driver should return the next error message in the buffer. When DataJoiner calls a different function, the generic driver should discard the current contents of the error message buffer.

The information returned by SQLError is in the format of SQLSTATE. For a list of error codes and the functions that return them, see Appendix A, “General Diagnostic Information” on page 89.

Constructing Error Messages

The error messages that the generic driver provides must include the SQLSTATE and corresponding error text. Error messages returned by SQLError come from two sources: data sources and the components in a generic driver connection. If a component in a generic driver connection receives an error message from a data source, it must identify the data source as the source of the error. It must also identify itself as the component that received the error.

The following are examples of two formats for the error text returned by SQLError: one for errors that occur in a data source and one for errors that occur in other components in the generic driver connection. For errors that do not occur in a data source, the error text format can be:

```
[vendor-identifier] component-supplied-text
```

For errors that do occur in a data source, the error text format can be:

```
[vendor-identifier] [data-source-identifier] data-source-supplied-text
```

Terminating Transactions and Connections

The generic access API allows termination of SQL transactions, statement-processing connections (*hstmt*s), connections (*hdbc*s), and environment connections (*henv*s).

Terminating Statement Processing

The SQLFreeStmt function releases resources associated with a statement handle. It has two options:

- SQL_CLOSE

  Close the cursor if one exists, and discard pending results. DataJoiner can use the statement handle again later.

- SQL_DROP

  Close the cursor, if one exists, discard pending results, and free all resources associated with the statement handle.

Terminating Transactions

The SQLTransact function requests a commit or rollback operation for the current transaction. The generic driver must submit a commit or rollback request for all operations associated with the specified *hdbc*; this includes operations for all *hstmt*s associated with *hdbc*.

Terminating Connections

To allow DataJoiner to terminate the connection to a generic driver and the data source, the generic driver must support the following three functions:

- SQLDisconnect

  Closes a connection. DataJoiner can then use the handle to reconnect to the same or a different data source.

- SQLFreeConnect

  Releases the connection handle and frees all resources associated with the handle.

- SQLFreeEnv

  Releases the environment handle and frees all resources associated with the handle.

Chapter 3. Required Functions

This chapter 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 explained below. The function descriptions follow immediately after.

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

Syntax
This section contains the 'C' prototype for the function under discussion.

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

Each argument is either an input argument or an output argument. The generic driver should modify only those arguments that are indicated as output.

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

Usage
This section provides information about how this function is used, and any special considerations. Possible error conditions are listed in the diagnostics section.

Return Codes
This section lists all the possible function return codes. When the generic driver returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, DataJoiner obtains error information by calling `SQLError()`.

Refer to “Returning Status and Error Information” on page 10 for more information about return codes.

Diagnostics

This section contains a table that lists the SQLSTATEs explicitly returned by the generic access API (SQLSTATEs generated by the DBMS can also be returned) and indicates the cause of the error. These values are obtained by calling `SQLError()` after the function returns a SQL_ERROR or SQL_SUCCESS_WITH_INFO.

Some SQLSTATEs are labeled *optional*. The errors that cause these optional SQLSTATEs should never occur. However, you can still test for their occurrence.

An “*****” in the first column indicates that the SQLSTATE is returned only by the generic access API, and will not be returned by other ODBC drivers.

Refer to “Returning Status and Error Information” on page 10 for more information about diagnostics. For a cross reference table, see Appendix A, “General Diagnostic Information” on page 89.

Restrictions

This section indicates any differences or limitations between the generic access API and ODBC that can affect DataJoiner.

Example

This section is a code fragment that demonstrates the use of the function.

References

This section lists related generic access API functions.

SQLAllocConnect - Allocate Connection Handle

Purpose

SQLAllocConnect() allocates a connection handle and associated resources within the environment identified by the input environment handle.

DataJoiner calls SQLAllocEnv() before calling this function. DataJoiner calls this function before calling SQLConnect() or SQLDriverConnect().

Syntax

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

Function Arguments

Table 4. SQLAllocConnect Arguments

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

Usage

The output connection handle is used by the generic access API to reference all information related to the connection, including information about the general status of the connection, the transaction state, and errors.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

If SQL_ERROR is returned, the *phdbc* argument will be set to SQL_NULL_HDBC. DataJoinerwill call SQLError() with the environment handle (*henv*) and with *hdbc* and *hstmt* arguments set to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.

Diagnostics

Table 5 (Page 1 of 2). SQLAllocConnect SQLSTATEs

CLI SQLSTATE	Description	Explanation
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1009 (optional)	Invalid argument value.	*phdbc* was a null pointer.

Table 5 (Page 2 of 2). SQLAllocConnect SQLSTATEs

CLI SQLSTATE	Description	Explanation
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.
S1014 *****	Out of handles.	A connection handle already exists for DataJoiner.

Restrictions

None.

Example

The following example shows how to obtain diagnostic information for the connection and the environment.

```
/*******************************************************************
** initialize
**  - allocate environment handle
**  - allocate connection handle
**  - prompt for server, user id, & password
**  - connect to server
*******************************************************************/

int initialize(SQLHENV *henv,
              SQLHDBC *hdbc)
{
SQLCHAR     server[SQL_MAX_DSN_LENGTH],
            uid[30],
            pwd[30];
SQLRETURN   rc;

    SQLAllocEnv (henv);         /* allocate an environment handle   */
    if (rc != SQL_SUCCESS )
        check_error (*henv, *hdbc, SQL_NULL_HSTMT, rc);

    SQLAllocConnect (*henv, hdbc);  /* allocate a connection handle     */
    if (rc != SQL_SUCCESS )
        check_error (*henv, *hdbc, SQL_NULL_HSTMT, rc);

    printf("Enter Server Name:\n");
    gets(server);
    printf("Enter User Name:\n");
    gets(uid);
    printf("Enter Password Name:\n");
    gets(pwd);

    if (uid[0] == '\0')
    {   rc = SQLConnect (*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
        if (rc != SQL_SUCCESS )
```

```
            check_error (*henv, *hdbc, SQL_NULL_HSTMT, rc);
    }
    else
    {   rc = SQLConnect (*hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
        if (rc != SQL_SUCCESS )
            check_error (*henv, *hdbc, SQL_NULL_HSTMT, rc);
    }
}/* end initialize */


/*******************************************************************/
int check_error (SQLHENV    henv,
                 SQLHDBC    hdbc,
                 SQLHSTMT   hstmt,
                 SQLRETURN  frc)
{
SQLRETURN   rc;

    print_error(henv, hdbc, hstmt);

    switch (frc){
    case SQL_SUCCESS : break;
    case SQL_ERROR :
    case SQL_INVALID_HANDLE:
        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");
        terminate(henv, hdbc);
        exit(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 ** \n");
        break;
    default :
        printf("\n ** Invalid Return Code ** \n");
        printf(" ** Attempting to rollback transaction **\n");
        SQLTransact(henv, hdbc, SQL_ROLLBACK);
        terminate(henv, hdbc);
        exit(frc);
        break;
    }
    return(SQL_SUCCESS);

}
```

SQLAllocConnect

References

- “SQLAllocEnv - Allocate Environment Handle” on page 19
- “SQLConnect - Connect to a Data Source” on page 27
- “SQLDisconnect - Disconnect from a Data Source” on page 35
- “SQLFreeConnect - Free Connection Handle” on page 55

SQLAllocEnv - Allocate Environment Handle

Purpose

SQLAllocEnv() allocates an environment handle and associated resources. There can only be one environment active at any one time per end user connection to DataJoiner.

DataJoiner calls this function prior to SQLAllocConnect() or any other generic access API functions. The *henv* value is passed in all subsequent function calls that require an environment handle as input.

Syntax

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

Function Arguments

Table 6. SQLAllocEnv Arguments

Data Type	Argument	Use	Description
SQLHENV *	*phenv*	output	Pointer to environment handle

Usage

There can be only one environment active at any one time per end user connection to DataJoiner. Any calls to SQLAllocEnv() will be rejected while a valid environment handle still exists.

As long as a valid environment handle is returned to DataJoiner, DataJoiner will issue a SQLFreeEnv() to invalidate the environment handle and free up the resources associated with the handle.

Return Codes

- SQL_SUCCESS
- SQL_ERROR

If SQL_ERROR is returned and *phenv* is equal to SQL_NULL_HENV, DataJoiner will not call SQLError() 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 SQLError() to obtain more error information, or to SQLFreeEnv().

Diagnostics

Table 7. SQLAllocEnv SQLSTATEs

SQLSTATE	Description	Explanation
58004	System error	The code page of the environment that DataJoiner is running in is not supported by the driver.

Restrictions

None.

Example

```
/******************************************************
** file = initterm.c
**    - demonstrate initialization and termination step.
**    - error handling has been ignored for simplicity.
**
**  Functions used:
**
**    SQLAllocConnect  SQLDisconnect
**    SQLAllocEnv      SQLFreeConnect
**    SQLConnect       SQLFreeEnv
**
*******************************************************/

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

int initialize(SQLHENV    *henv,
               SQLHDBC    *hdbc);

int terminate(SQLHENV     henv,
              SQLHDBC     hdbc);

#define MAX_UID_LENGTH    30
#define MAX_PWD_LENGTH    30

int main()
{
SQLHENV     henv;
SQLHDBC     hdbc;

    initialize(&henv, &hdbc);

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

    terminate(henv, hdbc);
    return (SQL_SUCCESS);
}

/********************************************************************/
int initialize(SQLHENV *henv,
               SQLHDBC *hdbc)
{
SQLRETURN   rc;
SQLCHAR     server[SQL_MAX_DSN_LENGTH + 1],
            uid[MAX_UID_LENGTH + 1],
            pwd[MAX_PWD_LENGTH + 1];

   printf("Enter Server Name:\n");
   gets(server);
   printf("Enter User Name:\n");
   gets(uid);
   printf("Enter Password Name:\n");
   gets(pwd);

   SQLAllocEnv (henv);             /* allocate an environment handle   */

   SQLAllocConnect (*henv, hdbc);  /* allocate a connection handle     */

   rc = SQLConnect (*hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
   if (rc != SQL_SUCCESS)
   {   printf("Error while connecting to database\n");
       return(SQL_ERROR);
   }
   else
       return (SQL_SUCCESS);
}

/********************************************************************/
int terminate(SQLHENV henv,
              SQLHDBC hdbc)
{
    SQLDisconnect (hdbc);          /* disconnect from database      */
    SQLFreeConnect (hdbc);         /* free connection handle        */
    SQLFreeEnv (henv);             /* free environment handle       */

    return(SQL_SUCCESS);
}
```

References

- “SQLAllocConnect - Allocate Connection Handle” on page 15
- “SQLFreeEnv - Free Environment Handle” on page 57

SQLAllocStmt - Allocate a Statement Handle

Purpose

SQLAllocStmt() allocates a new statement handle and associates it with the connection specified by the connection handle.

DataJoiner calls SQLConnect() or SQLDriverConnect() before calling this function.

DataJoiner calls this function before SQLSetParam(), SQLPrepare(), SQLExecute(), SQLExecDirect(), or any other function that has a statement handle as one of its input arguments.

Syntax

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

Function Arguments

Table 8. SQLAllocStmt Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle
SQLHSTMT	*phstmt* *	output	Pointer to statement handle

Usage

The generic access API uses each statement handle to relate all the descriptors, 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* reference an active database connection.

To execute a positioned update or delete, DataJoineruses different statement handles for the SELECT statement and the UPDATE or DELETE statement.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

If SQL_ERROR is returned, the *phstmt* argument will be set to SQL_NULL_HSTMT. DataJoiner will call SQLError() with the same *hdbc* and with the *hstmt* argument set to SQL_NULL_HSTMT.

Diagnostics

Table 9. SQLAllocStmt SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection not open.	The connection specified by the *hdbc* argument was not open. The connection must be established successfully (and the connection must be open) for the driver to allocate an *hstmt*.
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1009 (optional)	Invalid argument value.	*phstmt* was a null pointer.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.
S1014 *****	Out of handles.	No more statement handles are available for allocation.

Restrictions

None.

Example

Refer to “Example” on page 49.

References

- “SQLFreeStmt - Free (or Reset) a Statement Handle” on page 59

SQLBindCol - Bind a Column to DataJoiner Storage

Purpose

SQLBindCol() is used to associate (bind) DataJoiner storage buffers to columns in a result set. This enables data to be transferred from generic driver to DataJoiner when SQLFetch() is called. This function is also used to specify any data conversion required. It is called once for each column in the result set that DataJoinerneeds to retrieve.

DataJoiner usually calls SQLPrepare() or SQLExecDirect() before this function. It can also be necessary to call SQLDescribeCol().

DataJoiner calls SQLBindCol() before SQLFetch(), to transfer data to the storage buffers specified by this call.

Syntax

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

Function Arguments

Table 10 (Page 1 of 2). SQLBindCol Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLSMALLINT	*icol*	input	Number identifying the column. Columns are numbered sequentially, from left to right, starting at 1.
SQLSMALLINT	*fCType*	input	The C data type for column number *icol* in the result set. The following types are supported: • SQL_C_CHAR • SQL_C_DATE • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TIME • SQL_C_TIMESTAMP • SQL_C_DEFAULT Specifying SQL_C_DEFAULT causes data to be transferred to its default C data type. SQL_C_DEFAULT can not be specified for DECIMAL, NUMERIC or any of the graphic data types.
SQLPOINTER	*rgbValue*	output (deferred)	Pointer to buffer where the generic access API is to store the column data when the fetch occurs.

Table 10 (Page 2 of 2). SQLBindCol Arguments

Data Type	Argument	Use	Description
SQLINTEGER	*cbValueMax*	input	Size of *rgbValue* buffer in bytes available to store the column data. If *fCType* is either SQL_C_CHAR or SQL_C_DEFAULT, then *cbValueMax* must be > 0.
SQLINTEGER *	*pcbValue*	output (deferred)	Pointer to value which indicates the number of bytes the generic access API has available to return in the *rgbValue* buffer. `SQLFetch()` returns SQL_NULL_DATA in this argument if the data value of the column is null.

Note:

For this function, both *rgbValue* and *cbValue* are deferred outputs, meaning that the storage locations that these pointers point to do not get updated until `SQLFetch()` is called. As a result, the locations referenced by these pointers remain valid until `SQLFetch()` is called.

Usage

DataJoiner calls `SQLBindCol()` once for each column in the result set that it wants to retrieve. When `SQLFetch()` is called, the data in each of these *bound* columns is placed in the assigned location (given by the pointers *rgbValue* and *cbValue*).

DataJoiner can query the attributes (such as data type and length) of the column by first calling `SQLDescribeCol()`. This information can then be used to specify the correct data type of the storage locations, or to indicate data conversion to other data types.

Columns are identified by numbers assigned sequentially from left to right, starting at 1. The number of columns in the result set can be determined by calling `SQLNumResultCols()`.

DataJoiner will bind every column of the result set.

DataJoiner ensures that enough storage is allocated for the data to be retrieved. If the buffer is to contain variable length data, DataJoiner allocates as much storage as the maximum length of the bound column requires; otherwise, the data can be truncated. If data conversion is specified, the required size can be affected. See Appendix B, “Data Conversion” on page 95 for more information.

If string truncation does occur, SQL_SUCCESS_WITH_INFO should be returned and *pcbValue* will be set to the actual size of *rgbValue* available for return to DataJoiner.

If the column to be bound is a SQL_GRAPHIC, SQL_VARGRAPHIC or SQL_LONGVARGRAPHIC type, then *fCType* will be set to SQL_C_CHAR or an error will occur. Furthermore, the length of the *rgbValue* buffer (*cbValueMax*) will be a multiple of 2.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 11. SQLBindCol SQLSTATEs

SQLSTATE	Description	Explanation
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1002	Invalid column number. The value specified for the argument *icol* exceeded the maximum number of columns supported by the data source.	
S1003 (optional)	Program type out of range.	*fCType* was not a valid data type or SQL_C_DEFAULT.
S1009 (optional)	Invalid argument value.	The value specified for the argument *cbValueMax* is less than 1 and the argument *fCType* is either SQL_C_CHAR or SQL_C_DEFAULT.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.
S1C00	Driver not capable.	The driver recognizes, but does not support, the data type specified in the argument *fCType* (see also **S1**003).

Restrictions

None.

Example

Refer to “Example” on page 49.

References

- “SQLExecDirect - Execute a Statement Directly” on page 41
- “SQLExecute - Execute a Statement” on page 45
- “SQLFetch - Fetch Next Row” on page 47
- “SQLPrepare - Prepare a Statement” on page 73

SQLConnect - Connect to a Data Source

Purpose

SQLConnect() establishes a connection to the target database. DataJoiner supplies a target SQL database and, optionally, an authorization-name and an authentication-string. The database must be cataloged before DataJoiner can connect to it.

DataJoiner calls SQLAllocConnect() before calling this function.

DataJoiner calls this function before calling SQLAllocStmt().

Syntax

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

Function Arguments

Table 12. SQLConnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle
SQLCHAR *	*szDSN*	input	Data Source: The name or alias-name of the database.
SQLSMALLINT	*cbDSN*	input	Length of contents of *szDSN* argument
SQLCHAR *	*szUID*	input	Authorization-name (user identifier)
SQLSMALLINT	*cbUID*	input	Length of contents of *szUID* argument
SQLCHAR *	*szAuthStr*	input	Authentication-string (password)
SQLSMALLINT	*cbAuthStr*	input	Length of contents of *szAuthStr* argument

Usage

The target database (also known as *data source*) is the database-alias.

The data sources need to be cataloged only once.

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.

Leading and trailing blanks in the *szDSN* and *szUID* argument values should be stripped before processing unless they are enclosed in quotes.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 13. SQLConnect SQLSTATEs

SQLSTATE	Description	Explanation
08001	Unable to connect to data source.	The driver was unable to establish a connection with the data source (server).
08002 (optional)	Connection is used.	The specified *hdbc* has already been used to establish a connection with a data source and the connection is still open.
08004	Data source rejected establishment of connection.	The data source (server) rejected the establishment of the connection.
28000	Invalid authorization specification.	The value specified for the argument *szUID* or the value specified for the argument *szAuthStr* violated restrictions defined by the data source.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1009 (optional)	Invalid argument value.	The value specified for argument *cbDSN* was less than 0, but not equal to SQL_NTS and the argument *szDSN* was not a null pointer. The value specified for argument *cbUID* was less than 0, but not equal to SQL_NTS and the argument *szUID* was not a null pointer. The value specified for argument *cbAuthStr* was less than 0, but not equal to SQL_NTS and the argument *szAuthStr* was not a null pointer. A non-matching double quote (") was found in either the *szDSN*, *szUID*, or *szAuthStr* argument.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.
S1501 *****	Invalid data source name.	An invalid data source name was specified in argument *szDSN*.

Restrictions

DataJoiner calls `SQLConnect()` or `SQLDriverConnect()` before any SQL statements are executed.

Example

Refer to “Example” on page 20.

References

- “SQLAllocConnect - Allocate Connection Handle” on page 15
- “SQLAllocStmt - Allocate a Statement Handle” on page 22

SQLDescribeCol - Describe Column Attributes

Purpose

`SQLDescribeCol()` returns the result descriptor information (column name, type, precision) for the indicated column in the result set generated by a SELECT statement.

DataJoiner calls either `SQLPrepare()` or `SQLExecDirect()` before calling this function.

Syntax

```
SQLRETURN SQLDescribeCol (SQLHSTMT       hstmt,
                          SQLSMALLINT    icol,
                          SQLCHAR        *szColName,
                          SQLSMALLINT    cbColNameMax,
                          SQLSMALLINT    *pcbColName,
                          SQLSMALLINT    *pfSqlType,
                          SQLINTEGER     *pcbColDef,
                          SQLSMALLINT    *pibScale,
                          SQLSMALLINT    *pfNullable);
```

Function Arguments

Table 14 (Page 1 of 2). SQLDescribeCol Arguments

Data Type	**Argument**	**Use**	**Description**
SQLHSTMT	*hstmt*	input	Statement handle
SQLSMALLINT	*icol*	input	Column number to be described
SQLCHAR *	*szColName*	output	Pointer to column name buffer
SQLSMALLINT	*cbColNameMax*	input	Size of *szColName* buffer
SQLSMALLINT *	*pcbColName*	output	Bytes available to return for *szColName* argument. Truncation of column name (*szColName*) to *cbColNameMax - 1* bytes occurs if *pcbColName* is greater than or equal to *cbColNameMax*.
SQLSMALLINT *	*pfSqlType*	output	SQL data type of column.
SQLINTEGER *	*pcbColDef*	output	Precision of column as defined in the database. If *fSqlType* denotes a graphic SQL data type, then this variable indicates the maximum number of double-byte *characters* the column can hold.
SQLSMALLINT *	*pibScale*	output	Scale of column as defined in the database (only applies to SQL_DECIMAL, SQL_NUMERIC, SQL_TIMESTAMP). Refer to Table 53 on page 99 for the scale of each of the SQL datatypes.

Table 14 (Page 2 of 2). SQLDescribeCol Arguments

Data Type	Argument	Use	Description
SQLSMALLINT *	*pfNullable*	output	Indicates whether NULLS are allowed for this column • SQL_NO_NULLS • SQL_NULLABLE

Usage

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

A valid pointer and buffer space will be made available for the *szColName* argument. If a null pointer is specified for any of the remaining pointer arguments, the generic access API assumes that the information is not needed by DataJoiner and nothing is returned.

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 `SQLError()` function.

Table 15 (Page 1 of 2). SQLDescribeCol SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The column name returned in the argument *szColName* was longer than the value specified in the argument *cbColNameMax*. The argument *pcbColName* contains the length of the full column name. (Function returns SQL_SUCCESS_WITH_INFO.)
07005 *****	Not a SELECT statement.	The statement associated with the *hstmt* did not return a result set. There were no columns to describe. (Call `SQLNumResultCols()` first to determine if there are any rows in the result set.)
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.

Table 15 (Page 2 of 2). SQLDescribeCol SQLSTATEs

SQLSTATE	Description	Explanation
S1002 (optional)	Invalid column number.	The value specified for the argument *icol* was less than 1. The value specified for the argument *icol* was greater than the number of columns in the result set.
S1009 (optional)	Invalid argument value.	The length specified in argument *cbColNameMax* less than 1. The argument *szColName* was a null pointer.
S1010 (optional)	Function sequence error.	The function was called prior to calling `SQLPrepare()` or `SQLExecDirect()` for the *hstmt*.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.
S1C00	Driver not capable.	The SQL data type of column *icol* is not recognized by the generic access API.

Restrictions

None.

Example

```
/*******************************************************************
** file = typical.c
...
/*******************************************************************
** 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];
SQLSMALLINT     scale;
SQLINTEGER      outlen[MAXCOLS];
```

```
SQLCHAR *       data[MAXCOLS];
SQLCHAR         errmsg[256];
SQLRETURN       rc;
SQLINTEGER      i;
SQLINTEGER      displaysize;

    for (i = 0; i < nresultcols; i++)
    {
        SQLDescribeCol (hstmt, i+1, colname, sizeof (colname),
        &colnamelen, &coltype, &collen[i], &scale, NULL);

        /* get display length for column */
            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 */
        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),
                          "%d chars truncated, col %d\n",
                           outlen[i]-collen[i]+1, i+1);
            }
            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

- “SQLExecDirect - Execute a Statement Directly” on page 41
- “SQLNumResultCols - Get Number of Result Columns” on page 71
- “SQLPrepare - Prepare a Statement” on page 73

SQLDisconnect - Disconnect from a Data Source

Purpose

`SQLDisconnect()` closes the connection associated with the database connection handle.

DataJoiner calls `SQLTransact()` before calling `SQLDisconnect()` if an outstanding transaction exists on this connection.

After calling this function, DataJoiner calls either (1) `SQLConnect()` or `SQLDriverConnect()` to connect to another database, or (2) `SQLFreeConnect()`.

Syntax

```
SQLRETURN SQLDisconnect (SQLHDBC    hdbc);
```

Function Arguments

Table 16. SQLDisconnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle

Usage

If SQL_SUCCESS_WITH_INFO is returned, it implies that even though the disconnect from the database is successful, additional error-specific or implementation-specific information is available. Such information, for example, might be that a problem was encountered on the cleanup subsequent to the disconnect, or that there is no current connection because of an event that occurred independently of DataJoiner (such as communication failure).

After a successful `SQLDisconnect()` call, DataJoinercan reuse *hdbc* to make another `SQLConnect()` or `SQLDriverConnect()` request.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 17. SQLDisconnect SQLSTATEs

SQLSTATE	Description	Explanation
01002	Disconnect error.	An error occurred during the disconnect. However, the disconnect succeeded. (Function returns SQL_SUCCESS_WITH_INFO.)
08003 (optional)	Connection not open.	The connection specified in the argument *hdbc* was not open.
25000 (optional)	Invalid transaction state.	There was a transaction in process on the connection specified by the argument *hdbc*. The transaction remains active, and the connection cannot be disconnected.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 20.

References

- “SQLAllocConnect - Allocate Connection Handle” on page 15
- “SQLConnect - Connect to a Data Source” on page 27
- “SQLTransact - Transaction Management” on page 87

SQLError - Retrieve Error Information

Purpose

SQLError() returns the diagnostic information associated with the most recently invoked generic access API function for a particular statement, connection or environment handle.

The information consists of a standardized SQLSTATE, native error code, and a text message. Refer to “Returning Status and Error Information” on page 10 for more information.

DataJoiner calls SQLError() after receiving a return code of SQL_ERROR or SQL_SUCCESS_WITH_INFO from another function call.

Note: Some database servers can provide product-specific diagnostic information after returning SQL_NO_DATA_FOUND from the execution of a statement.

Syntax

```
SQLRETURN SQLError (SQLHENV      henv,
                    SQLHDBC      hdbc,
                    SQLHSTMT     hstmt,
                    SQLCHAR      *szSqlState,
                    SQLINTEGER   *pfNativeError,
                    SQLCHAR      *szErrorMsg,
                    SQLSMALLINT  cbErrorMsgMax,
                    SQLSMALLINT  *pcbErrorMsg);
```

Function Arguments

Table 18 (Page 1 of 2). SQLError Arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle. To obtain diagnostic information associated with an environment, DataJoiner will pass a valid environment handle. It will also set *hdbc* and *hstmt* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
SQLHDBC	*hdbc*	input	Database connection handle. To obtain diagnostic information associated with a connection, DataJoiner will also pass a valid database connection handle, and set *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored.
SQLHSTMT	*hstmt*	input	Statement handle. To obtain diagnostic information associated with a statement, DataJoiner will pass a valid statement handle. The *henv* and *hdbc* arguments are ignored.

Table 18 (Page 2 of 2). SQLError Arguments

Data Type	Argument	Use	Description
SQLCHAR *	*szSqlState*	output	SQLSTATE as a string of 5 characters terminated by a null character. The first 2 characters indicate error class; the next 3 indicate subclass.
SQLINTEGER *	*pfNativeError*	output	Native error code. In the generic access API, the *pfNativeError* argument will contain the SQLCODE value returned by the DBMS. If the error is generated by the generic access API and not the DBMS, then this field will be set to -99999.
SQLCHAR *	*szErrorMsg*	output	Pointer to buffer to contain the implementation defined message text. In the generic access API, only the DBMS generated messages will be returned; the generic access API itself will not return any message text describing the problem.
SQLSMALLINT *	*pcbErrorMsg*	output	Pointer to total number of bytes available to return to the *szErrorMsg* buffer. This does not include the null termination character.

Usage

The SQLSTATEs are those defined by the X/OPEN SQL and the X/Open SQL CLI snapshot.

To obtain diagnostic information associated with an environment, DataJoiner passes a valid environment handle and sets *hdbc* and *hstmt* to SQL_NULL_HDBC and SQL_NULL_HSTMT, respectively. To obtain diagnostic information associated with a connection, DataJoiner passes a valid database connection handle and sets *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored. To obtain diagnostic information associated with a statement, DataJoiner passes a valid statement handle. The *henv* and *hdbc* arguments are ignored.:

If diagnostic information generated by one generic access API 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 generic access API function call.

Multiple diagnostic messages can be available after a given generic access API function call. These messages are retrieved one at a time by repeatedly calling `SQLError()`. For each message retrieved, `SQLError()` returns SQL_SUCCESS and removes it from the list of messages available. When there are no more messages to retrieve, SQL_NO_DATA_FOUND should return, the SQLSTATE is set to “00000,” *pfNativeError* is set to 0, and *pcbErrorMsg* and *szErrorMsg* are undefined.

Diagnostic information stored under a given handle is cleared when a call is made to `SQLError()` with that handle, or when another generic access API function call is made with that handle. However, information associated with a given handle type is not cleared by a call to `SQLError()` with an associated but different handle type: for example, a call to SQLError() with a connection handle input will not clear errors associated with any statement handles under that connection.

SQL_SUCCESS should be returned even if the buffer for the error message (*szErrorMsg*) is too short since DataJoinerwill not be able to retrieve the same error message by calling SQLError() again. The actual length of the message text is returned in the *pcbErrorMsg*.

To avoid truncation of the error message, DataJoinerdeclares a buffer length of SQL_MAX_MESSAGE_LENGTH + 1. The message text should never be longer than this.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND should be returned if no diagnostic information is available for the input handle, or if all of the messages have been retrieved via calls to SQLError().

Diagnostics

SQLSTATEs are not defined, since SQLError() does not generate diagnostic information for itself.

Restrictions

None.

Example

```
/*************************************************************************
** file = typical.c
***********************************************************************/
int print_error (SQLHENV    henv,
                    SQLHDBC    hdbc,
                    SQLHSTMT   hstmt)
{
SQLCHAR     buffer[SQL_MAX_MESSAGE_LENGTH + 1];
SQLCHAR     sqlstate[SQL_SQLSTATE_SIZE + 1];
SQLINTEGER  sqlcode;
SQLSMALLINT length;


    while ( SQLError(henv, hdbc, hstmt, sqlstate, &sqlcode, buffer,
                     SQL_MAX_MESSAGE_LENGTH + 1, &length) == SQL_SUCCESS )
    {
        printf("\n **** ERROR *****\n");
        printf("        SQLSTATE: %s\n", sqlstate);
        printf("Native Error Code: %ld\n", sqlcode);
        printf("%s \n", buffer);
    };
    return (0);

}
```

References

None.

SQLExecDirect - Execute a Statement Directly

Purpose

SQLExecDirect() directly executes the specified SQL statement. The statement can be executed only once. Also, the connected database server must be able to prepare the statement.

Syntax

```
SQLRETURN SQLExecDirect (SQLHSTMT      hstmt,
                         SQLCHAR       *szSqlStr,
                         SQLINTEGER    cbSqlStr);
```

Function Arguments

Table 19. SQLExecDirect Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle. There must not be an open cursor associated with *hstmt*.
SQLCHAR *	*szSqlStr*	input	SQL statement string. The connected database server must be able to prepare the statement.
SQLINTEGER	*cbSqlStr*	input	Length of contents of *szSqlStr* argument. The length will be set to either the exact length of the statement, or if the statement is null-terminated, set to SQL_NTS.

Usage

The SQL statement cannot be a COMMIT or ROLLBACK. Instead, SQLTransact() must be called to issue COMMIT or ROLLBACK.

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 the value of DataJoiner storage is to be substituted, when SQLExecDirect() is called. SQLSetParam() is used to bind (or associate) DataJoiner storage to each parameter marker, and to indicate if any data conversion should be performed at the time the data is transferred. All parameters will be bound before calling SQLExecDirect().

If the SQL statement is a SELECT, SQLExecDirect() will generate a cursor name and open the cursor.

To retrieve a row from the result set generated by a SELECT statement, DataJoiner calls SQLFetch() after SQLExecDirect() returns successfully.

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.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND should be returned if the SQL statement is a Searched UPDATE or Searched DELETE and no rows satisfy the search condition.

Diagnostics

Table 20 (Page 1 of 3). SQLExecDirect SQLSTATEs

SQLSTATE	Description	Explanation
01504 *****	No WHERE clause.	*szSqlStr* contained an UPDATE or DELETE statement which did not contain a WHERE clause. (Function returns SQL_SUCCESS_WITH_INFO or SQL_NO_DATA_FOUND if there were no rows in the table).
01508 *****	Statement disqualified for blocking.	The statement was disqualified for blocking for reasons other than storage.
07001 (optional)	Wrong number of parameters.	The number of parameters bound to DataJoiner storage using `SQLSetParam()` was less than the number of parameter marker in the SQL statement contained in the argument *szSqlStr*.
07006	Restricted data type attribute violation.	Transfer of data between the generic access API and DataJoiner storage would result in incompatible data conversion.
21S01	Insert value list does not match column list.	*szSqlStr* contained an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*szSqlStr* contained a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
22001	String data right truncation.	A character string assigned to a character type column exceeded the maximum length of the column.
22003 (optional)	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. *szSqlStr* contained an SQL statement with an arithmetic expression which caused division by zero. **Note:** As a result DataJoiner will treat the cursor state as if it were undefined.

Table 20 (Page 2 of 3). SQLExecDirect SQLSTATEs

SQLSTATE	Description	Explanation
22005	Error in assignment.	*szSqlStr* contained an SQL statement with a parameter or literal and the value was incompatible with the data type of the associated table column. The argument *fSQLType* used in `SQLSetParam()` denoted an SQL graphic data type, but the deferred length argument (*pcbValue*) contains an odd length value. The length value must be even for graphic data types.
22007 *****	Invalid date time format.	*szSqlStr* contained an SQL statement with an invalid datetime format; that is, an invalid string representation or value was specified, or the value was an invalid date.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero.	*szSqlStr* contained an SQL statement with an arithmetic expression that caused division by zero.
23000	Integrity constraint violation.	The execution of the SQL statement is not permitted because the execution would cause integrity constraint violation in the DBMS.
24000 (optional)	Invalid cursor state.	The cursor is not in the appropriate state for the execution of this SQL statement.
24504 ***** (optional)	Invalid cursor state.	Results were pending on the *hstmt* from a previous SELECT statement or a cursor associated with the *hsmt* had not been closed.
34000 (optional)	Invalid cursor name.	*szSqlStr* contained a Positioned DELETE or a Positioned UDPATE and the cursor referenced by the statement being executed was not open.
37xxx	Syntax error or access violation.	*szSqlStr* contained one or more of the following: • a COMMIT • a ROLLBACK • an SQL statement that the connected database server could not prepare • a statement containing a syntax error
40000 *****	Serialization failure.	The transaction to which this SQL statement belonged was rolled back due to a deadlock or timeout.
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
42504	Syntax error or access violation.	The current user did not have permission to execute the SQL statement contained in *szSqlStr*.

Table 20 (Page 3 of 3). SQLExecDirect SQLSTATEs

SQLSTATE	Description	Explanation
44000	Integrity constraint violation.	*szSqlStr* contained an SQL statement which contained a parameter or literal. This parameter value was NULL for a column defined as NOT NULL in the associated table column, or a duplicate value was supplied for a column constrained to contain only unique values, or some other integrity constraint was violated.
58004	System error.	Unrecoverable system error.
S0001	Base table or view already exists.	*szSqlStr* contained a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already existed.
S0002	Table or view not found.	*szSqlStr* contained an SQL statement that references a table name or view name which does not exist.
S0011	Index already exists.	*szSqlStr* contained a CREATE INDEX statement and the specified index name already existed.
S0012	Index not found.	*szSqlStr* contained a DROP INDEX statement and the specified index name did not exist.
S0021	Column already exists.	*szSqlStr* contained an ALTER TABLE statement and the column specified in the ADD clause was not unique or identified an existing column in the base table.
S0022	Column not found.	*szSqlStr* contained an SQL statement that references a column name which does not exist.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1009 (optional)	Invalid argument value.	*szSqlStr* was a null pointer. The argument *cbSqlStr* was less than 1 but not equal to SQL_NTS.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 49.

References

- “SQLExecute - Execute a Statement” on page 45
- “SQLFetch - Fetch Next Row” on page 47
- “SQLSetParam - Set Parameter” on page 82

SQLExecute - Execute a Statement

Purpose

SQLExecute() executes a statement that was successfully prepared using SQLPrepare (), once or multiple times. The statement is executed using the current values of any DataJoiner storage locations that were bound to parameter markers by SQLSetParam().

Syntax

```
SQLRETURN SQLExecute (SQLHSTMT     hstmt);
```

Function Arguments

Table 21. SQLExecute Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle. There must not be an open cursor associated with hstmt.

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 the value of DataJoiner storage is to be substituted, when SQLExecute() is called. SQLSetParam() is used to bind (or associate) DataJoiner storage to each parameter marker, and to indicate if any data conversion should be performed at the time that the data is transferred. All parameters must be bound before calling SQLExecute().

Once DataJoiner has processed the results from the SQLExecute() call, it can execute the statement again with new (or the same) values in the DataJoiner storage.

A statement executed by SQLExecDirect() cannot be re-executed by calling SQLExecute(); SQLPrepare() must be called first.

If the prepared SQL statement is a SELECT, SQLExecute() will generate a cursor name and open the cursor.

To execute a SELECT statement more than once, DataJoinercloses the cursor by calling call SQLFreeStmt() with the SQL_CLOSE option. There must not be an open cursor on the statement handle when calling SQLExecute().

To retrieve a row from the result set generated by a SELECT statement, DataJoiner calls SQLFetch() after SQLExecute() returns successfully.

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 that SQLExecute()

is called, and must be defined on a separate statement handle under the same connection handle.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND should be returned if the SQL statement is a Searched UPDATE or Searched DELETE and no rows satisfy the search condition.

Diagnostics

The SQLSTATEs for `SQLExecute()` include all those for `SQLExecDirect()` (refer to Table 20 on page 42) except for **S1**009, and with the addition of the SQLSTATE in the table below.

Table 22. SQLExecute SQLSTATEs

SQLSTATE	Description	Explanation
S1010 (optional)	Function sequence error.	The specified *hstmt* was not in prepared state. `SQLExecute()` was called without first calling `SQLPrepare`.

Note: `SQLExecute()` also returns the states in Table 20 on page 42, except for **S1**009.

Restrictions

None.

Example

Refer to “Example” on page 75.

References

- “SQLExecDirect - Execute a Statement Directly” on page 41
- “SQLBindCol - Bind a Column to DataJoiner Storage” on page 24
- “SQLPrepare - Prepare a Statement” on page 73
- “SQLFetch - Fetch Next Row” on page 47
- “SQLSetParam - Set Parameter” on page 82

SQLFetch - Fetch Next Row

Purpose

SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns.

SQLFetch() is used to receive the data directly into DataJoiner storage specified with SQLBindCol(). Data conversion is also performed when SQLFetch() is called, if conversion was indicated when the column was bound.

Syntax

```
SQLRETURN SQLFetch (SQLHSTMT    hstmt);
```

Function Arguments

Table 23. SQLFetch Arguments

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

Usage

SQLFetch() can only be called if the most recently executed statement on *hstmt*, was a SELECT.

The number of storage locations bound to DataJoiner with SQLBindCol() will not exceed the number of columns in the result set or SQLFetch() will fail.

If SQLBindCol() has not been called to bind any columns, then SQLFetch() does not return data to DataJoiner, but just advances the cursor. Data in unbound columns is discarded when SQLFetch() advances the cursor to the next row.

If any bound variables are not large enough to hold the data returned by SQLFetch(), the data will be truncated. If character data is truncated, SQL_SUCCESS_WITH_INFO should be returned, and an SQLSTATE is generated indicating truncation. The SQLBindCol() deferred output argument *pcbValue* will contain the actual length of the column data retrieved from the server. DataJoiner compares the output length to the input length (*pcbValue* and *cbValueMax* arguments from SQLBindCol()) to determine which character columns have been truncated.

Truncation of numeric data types is not reported if the truncation involves digits to the right of the decimal point. If truncation occurs to the left of the decimal point, an error should be returned (refer to the diagnostics section).

Truncation of graphic data types is treated the same as character data types, except that the *rgbValue* buffer is filled to the nearest multiple of two bytes that is still less than or equal to the *cbValueMax* specified in SQLBindCol().

When all the rows have been retrieved from the result set, or the remaining rows are not needed, `SQLFreeStmt()` will be called to close the cursor and discard the remaining data and associated resources.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND should be returned if there are no rows in the result set, or if previous `SQLFetch()` calls have fetched all the rows from the result set.

Diagnostics

Table 24 (Page 1 of 2). SQLFetch SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data returned for one or more columns was truncated. String values are right truncated. (SQL_SUCCESS_WITH_INFO is should be returned if no error occurred.)
07002 ***** (optional)	Invalid column number.	A column number specified in the binding for one or more columns was greater than the number of columns in the result set.
07006	Restricted data type attribute violation.	The data value could not be converted to the data type specified by *fCType* in SQLBindCol. A call to SQLBindCol was made with a value of SQL_C_DEFAULT for the argument *fCType* and the SQL data type of the corresponding column is one of SQL_DECIMAL, SQL_NUMERIC, SQL_GRAPHIC, SQL_VARGRAPHIC, or SQL_LONGVARGRAPHIC.
22002 (optional)	Invalid length buffer.	The pointer value specified for the argument pcbValue in SQLBindCol was a null pointer and the value of the corresponding column is null. There is no means to report SQL_NULL_DATA.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for one or more columns would have caused the whole part of the number to be truncated either at the time of assignment or in computing an intermediate result. A value from an arithmetic expression was returned which resulted in division by zero. **Note:** As a result, DataJoiner will treat the cursor as if it were undefined.
22005	Error in assignment.	A returned value was incompatible with the data type of binding.
22007 *****	Invalid date time format.	szSqlStr contained an SQL statement with an invalid datetime format; that is, an invalid string representation or value was specified, or the value was an invalid date.

Table 24 (Page 2 of 2). SQLFetch SQLSTATEs

SQLSTATE	Description	Explanation
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero.	A value from an arithmetic expression was returned which resulted in division by zero.
24000 (optional)	Invalid cursor state.	The previous SQL statement executed on the *hstmt* was not a SELECT.
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1010 (optional)	Function sequence error.	The specified *hstmt* was not in an executed state. The function was called without first calling SQLExecute or SQLExecDirect.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.
S1C00	Driver not capable.	The driver or data source does not support the conversion specified by the combination of the *fCType* 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 the driver.

Restrictions

None.

Example

```
/*************************************************************************
** file = fetch.c
**
** Example of executing an SQL statement.
** SQLBindCol & SQLFetch is used to retrieve data from the result set
** directly into application storage.
**
** Functions used:
**
**        SQLAllocConnect      SQLFreeConnect
**        SQLAllocEnv          SQLFreeEnv
**        SQLAllocStmt         SQLFreeStmt
**        SQLConnect           SQLDisconnect
**
```

```
**        SQLBindCol           SQLFetch
**        SQLTransact          SQLExecDirect
**        SQLError
**
**************************************************************************/

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

#define MAX_STMT_LEN 255

int initialize(SQLHENV *henv,
              SQLHDBC *hdbc);

int terminate(SQLHENV henv,
              SQLHDBC hdbc);

int print_error (SQLHENV    henv,
                 SQLHDBC    hdbc,
                 SQLHSTMT   hstmt);

int check_error (SQLHENV    henv,
                 SQLHDBC    hdbc,
                 SQLHSTMT   hstmt,
                 SQLRETURN  frc);

/*******************************************************************
** main
** - initialize
** - terminate
*******************************************************************/
int main()
{
    SQLHENV     henv;
    SQLHDBC     hdbc;
    SQLCHAR     sqlstmt[MAX_STMT_LEN + 1]="";
    SQLRETURN   rc;

    rc = initialize(&henv, &hdbc);
    if (rc == SQL_ERROR) return(terminate(henv, hdbc));


    {SQLHSTMT   hstmt;
     SQLCHAR    sqlstmt[]="SELECT deptname, location from org
           where division = 'Eastern'";
     SQLCHAR    deptname[15],
                location[14];
     SQLINTEGER rlength;

        rc = SQLAllocStmt(hdbc, &hstmt);
        if (rc != SQL_SUCCESS )
```

```
           check_error (henv, hdbc, SQL_NULL_HSTMT, rc);

        rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt, rc);

        rc = SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) deptname, 15,
                        &rlength);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt, rc);
        rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) location, 14,
                        &rlength);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt, rc);

        printf("Departments in Eastern division:\n");
        printf("DEPTNAME       Location\n");
        printf("-------------- -------------\n");

        while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
        {
            printf("%-14.14s %-13.13s \n", deptname, location);
        }
        if (rc != SQL_NO_DATA_FOUND )
           check_error (henv, hdbc, hstmt, rc);

        rc = SQLFreeStmt(hstmt, SQL_DROP);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, SQL_NULL_HSTMT, rc);
    }


    rc = SQLTransact(henv, hdbc, SQL_COMMIT);
    if (rc != SQL_SUCCESS )
        check_error (henv, hdbc, SQL_NULL_HSTMT, rc);

    terminate(henv, hdbc);
    return (0);
}/* end main */

/*******************************************************************
** initialize
**  - allocate environment handle
**  - allocate connection handle
**  - prompt for server, user id, & password
**  - connect to server
*******************************************************************/

int initialize(SQLHENV *henv,
              SQLHDBC *hdbc)
{
SQLCHAR     server[SQL_MAX_DSN_LENGTH],
            uid[30],
            pwd[30];
SQLRETURN   rc;

    rc = SQLAllocEnv (henv);         /* allocate an environment handle    */
    if (rc != SQL_SUCCESS )
        check_error (*henv, *hdbc, SQL_NULL_HSTMT, rc);

    rc = SQLAllocConnect (*henv, hdbc);  /* allocate a connection handle   */
    if (rc != SQL_SUCCESS )
        check_error (*henv, *hdbc, SQL_NULL_HSTMT, rc);

    printf("Enter Server Name:\n");
    gets(server);
    printf("Enter User Name:\n");
    gets(uid);
    printf("Enter Password Name:\n");
    gets(pwd);

    if (uid[0] == '\0')
    {   rc = SQLConnect (*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
        if (rc != SQL_SUCCESS )
            check_error (*henv, *hdbc, SQL_NULL_HSTMT, rc);
    }
    else
    {   rc = SQLConnect (*hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
        if (rc != SQL_SUCCESS )
            check_error (*henv, *hdbc, SQL_NULL_HSTMT, rc);
    }

    return(SQL_SUCCESS);
}/* end initialize */

/*******************************************************************
** terminate
**  - disconnect
**  - free connection handle
**  - free environment handle
*******************************************************************/
int terminate(SQLHENV henv,
              SQLHDBC hdbc)
{
SQLRETURN   rc;

    rc = SQLDisconnect (hdbc);               /* disconnect from database  */
    if (rc != SQL_SUCCESS )
        print_error (henv, hdbc, SQL_NULL_HSTMT);
    rc = SQLFreeConnect (hdbc);              /* free connection handle    */
    if (rc != SQL_SUCCESS )
        print_error (henv, hdbc, SQL_NULL_HSTMT);
    rc = SQLFreeEnv (henv);                  /* free environment handle   */
    if (rc != SQL_SUCCESS )
```

```
        print_error (henv, hdbc, SQL_NULL_HSTMT);

    return(rc);
}/* end terminate */

/*******************************************************************
**  - print_error   - call SQLError(), display SQLSTATE and message
*******************************************************************/

int print_error (SQLHENV    henv,
                 SQLHDBC    hdbc,
                 SQLHSTMT   hstmt)
{
SQLCHAR     buffer[SQL_MAX_MESSAGE_LENGTH + 1];
SQLCHAR     sqlstate[SQL_SQLSTATE_SIZE + 1];
SQLINTEGER  sqlcode;
SQLSMALLINT length;


    while ( SQLError(henv, hdbc, hstmt, sqlstate, &sqlcode, buffer,
                     SQL_MAX_MESSAGE_LENGTH + 1, &length) == SQL_SUCCESS )
    {
        printf("\n **** ERROR *****\n");
        printf("         SQLSTATE: %s\n", sqlstate);
        printf("Native Error Code: %ld\n", sqlcode);
        printf("%s \n", buffer);
    };

    return ( SQL_ERROR);
} /* end print_error */

/*******************************************************************
**  - check_error   - call print_error(), checks severity of return code
*******************************************************************/
int check_error (SQLHENV    henv,
                 SQLHDBC    hdbc,
                 SQLHSTMT   hstmt,
                 SQLRETURN  frc)
{
SQLRETURN   rc;

    print_error(henv, hdbc, hstmt);

    switch (frc){
    case SQL_SUCCESS : break;
    case SQL_ERROR :
    case SQL_INVALID_HANDLE:
        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");
        terminate(henv, hdbc);
        exit(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 ** \n");
        break;
    default :
        printf("\n ** Invalid Return Code ** \n");
        printf(" ** Attempting to rollback transaction **\n");
        SQLTransact(henv, hdbc, SQL_ROLLBACK);
        terminate(henv, hdbc);
        exit(frc);
        break;
    }
    return(SQL_SUCCESS);

} /* end check_error */
```

References

- “SQLBindCol - Bind a Column to DataJoiner Storage” on page 24
- “SQLExecute - Execute a Statement” on page 45
- “SQLExecDirect - Execute a Statement Directly” on page 41

SQLFreeConnect - Free Connection Handle

Purpose

SQLFreeConnect() invalidates and frees the connection handle. All the generic access API resources associated with the connection handle are freed.

DataJoiner calls SQLDisconnect() before calling this function.

Next, DataJoiner calls either SQLFreeEnv() to continue terminating the DataJoiner connection, or SQLAllocHandle(), to allocate a new connection handle.

Syntax

```
SQLRETURN SQLFreeConnect (SQLHDBC    hdbc);
```

Function Arguments

Table 25. SQLFreeConnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle

Usage

If this function is called when a connection still exists, SQL_ERROR should be returned, and the connection handle remains valid.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 26. SQLFreeConnect SQLSTATEs

SQLSTATE	Description	Explanation
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1010 (optional)	Function sequence error.	The function was called prior to SQLDisconnect() for the *hdbc*.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 20.

References

- “SQLDisconnect - Disconnect from a Data Source” on page 35
- “SQLFreeEnv - Free Environment Handle” on page 57

SQLFreeEnv - Free Environment Handle

Purpose

SQLFreeEnv() invalidates and frees the environment handle. All the generic access API resources associated with the environment handle are freed.

DataJoiner calls SQLFreeConnect() before calling this function.

This function is the last step that DataJoiner needs to perform before terminating the connection.

Syntax

```
SQLRETURN SQLFreeEnv (SQLHENV    henv);
```

Function Arguments

Table 27. SQLFreeEnv Arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle

Usage

If this function is called when there is still a valid connection handle, SQL_ERROR should be returned, and the environment handle will remain valid.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 28. SQLFreeEnv SQLSTATEs

SQLSTATE	Description	Explanation
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1010 (optional)	Function sequence error.	There is an *hdbc* which is in allocated or connected state. Call SQLDisconnect and SQLFreeConnect for the *hdbc* before calling SQLFreeEnv.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 20.

References

- “SQLFreeConnect - Free Connection Handle” on page 55

SQLFreeStmt - Free (or Reset) a Statement Handle

Purpose

SQLFreeStmt() ends processing on the statement referenced by the statement handle. This function is used to close a cursor, or to drop the statement handle and free the generic access API resources associated with the statement handle.

DataJoiner calls SQLFreeStmt() after executing an SQL statement and processing the results.

Syntax

```
SQLRETURN SQLFreeStmt (SQLHSTMT       hstmt,
                       SQLSMALLINT    fOption);
```

Function Arguments

Table 29. SQLFreeStmt Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLSMALLINT	*fOption*	input	Option which specified the manner of freeing the statement handle. The option must have one of the following values: • SQL_CLOSE • SQL_DROP

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. DataJoiner can reopen the cursor by calling SQLExecute() with the same or different values in DataJoiner storage locations (if any) that are bound to *hstmt*. The cursor name is retained until the statement handle is dropped. If no cursor has been associated with the statement handle, this option has no effect (no warning or error is generated).

SQL_DROP DataJoiner's Generic Access API 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.

If you want to reuse a statement handle to execute a different statement, and if the previous statement was a SELECT, you must close the cursor. 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 `SQLError()` is called.

Diagnostics

Table 30. SQLFreeStmt SQLSTATEs

SQLSTATE	Description	Explanation
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	The value specified for the argument *fOption* was not SQL_CLOSE, or SQL_DROP.

Restrictions

None.

Example

Refer to “Example” on page 49.

References

- “SQLAllocStmt - Allocate a Statement Handle” on page 22
- “SQLBindCol - Bind a Column to DataJoiner Storage” on page 24
- “SQLFetch - Fetch Next Row” on page 47
- “SQLFreeConnect - Free Connection Handle” on page 55
- “SQLSetParam - Set Parameter” on page 82

SQLGetCursorName - Get Cursor Name

Purpose

SQLGetCursorName() returns the cursor name associated with the input statement handle.

Syntax

```
SQLRETURN SQLGetCursorName (SQLHSTMT      hstmt,
                            SQLCHAR       *szCursor,
                            SQLSMALLINT   cbCursorMax,
                            SQLSMALLINT   *pcbCursor);
```

Function Arguments

Table 31. SQLGetCursorName Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLCHAR *	*szCursor*	output	Cursor name
SQLSMALLINT	*cbCursorMax*	input	Length of buffer *szCursor*
SQLSMALLINT *	*pcbCursor*	output	Amount of bytes available to return for *szCursor*

Usage

SQLGetCursorName() will return a cursor name only if a SELECT statement was executed on the statement handle. Otherwise, calling SQLGetCusorName() will result in an error.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 32 (Page 1 of 2). SQLGetCursorName SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The cursor name returned in *szCursor* was longer than the value in *cbCursorMax*, and is truncated to *cbCursorMax* - 1 bytes. The argument *pcbCursor* contains the length of the full cursor name available for return. The function returns SQL_SUCCESS_WITH_INFO.

Table 32 (Page 2 of 2). SQLGetCursorName SQLSTATEs

SQLSTATE	Description	Explanation
40003 *	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1009 (optional)	Invalid argument value.	szCursor was a null pointer. The value specified for the argument *cbCursorMax* is less than 1.
S1010 (optional)	Function sequence error.	The statement *hstmt* is not in execute state. Call `SQLExecute()` or `SQLExecDirect()` before calling `SQLGetCursorName()`.
S1013 *	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.
S1015	No cursor name available.	There was no open cursor on the *hstmt*. The statement associated with *hstmt* does not support the use of a cursor.

Restrictions

None.

Example

```
/*************************************************************************
** file = getcurs.c
**
** Example of directly executing a SELECT and positioned UPDATE SQL statement.
** Two statement handles are used, and SQLGetCursor is used to retrieve the
** generated cursor name.
**
** Functions used:
**
**       SQLAllocConnect       SQLFreeConnect
**       SQLAllocEnv           SQLFreeEnv
**       SQLAllocStmt          SQLFreeStmt
**       SQLConnect            SQLDisconnect
**
**       SQLBindCol            SQLFetch
**       SQLTransact           SQLError
**       SQLExecDirect         SQLGetCursorName
**************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sqlcli1.h"

#define MAX_STMT_LEN 255
```

```
int initialize(SQLHENV *henv,
               SQLHDBC *hdbc);

int terminate(SQLHENV henv,
              SQLHDBC hdbc);

int print_error (SQLHENV    henv,
                 SQLHDBC    hdbc,
                 SQLHSTMT   hstmt);

int check_error (SQLHENV    henv,
                 SQLHDBC    hdbc,
                 SQLHSTMT   hstmt,
                 SQLRETURN  frc);

/*******************************************************************
** main
** - initialize
** - terminate
*******************************************************************/
int main()
{
    SQLHENV     henv;
    SQLHDBC     hdbc;
    SQLRETURN   rc,
                rc2;

    rc = initialize(&henv, &hdbc);
    if (rc != SQL_SUCCESS) return(terminate(henv, hdbc));

    {SQLHSTMT   hstmt1,
                hstmt2;
     SQLCHAR    sqlstmt[]="SELECT name, job from staff for update of job";
     SQLCHAR    updstmt[MAX_STMT_LEN + 1];
     SQLCHAR    name[10],
                job[6],
                newjob[6],
                cursor[19];

     SQLINTEGER     rlength;
     SQLSMALLINT    clength;

        rc = SQLAllocStmt(hdbc, &hstmt1);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, SQL_NULL_HSTMT, rc);

        /* allocate second statement handle for update statement */
        rc2 = SQLAllocStmt(hdbc, &hstmt2);
        if (rc2 != SQL_SUCCESS )
            check_error (henv, hdbc, SQL_NULL_HSTMT, rc);
        rc = SQLExecDirect(hstmt1, sqlstmt, SQL_NTS);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt1, rc);

        /* Get Cursor of the SELECT statement's handle */
        rc = SQLGetCursorName(hstmt1, cursor, 19, &clength);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt1, rc);

        /* bind name to first column in the result set */
        rc = SQLBindCol(hstmt1, 1, SQL_C_CHAR, (SQLPOINTER) name, 10,
                        &rlength);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt1, rc);

        /* bind job to second column in the result set */
        rc = SQLBindCol(hstmt1, 2, SQL_C_CHAR, (SQLPOINTER) job, 6,
                         &rlength);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt1, rc);


        printf("Job Change for all clerks\n");

        while ((rc = SQLFetch(hstmt1)) == SQL_SUCCESS)
        {
            printf("Name: %-9.9s Job: %-5.5s \n", name, job);
            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 %s",
                    newjob, cursor);
                rc2 = SQLExecDirect(hstmt2, updstmt, SQL_NTS);
                if (rc2 != SQL_SUCCESS )
                    check_error (henv, hdbc, hstmt2, rc);
            }
        }
        if (rc != SQL_NO_DATA_FOUND )
            check_error (henv, hdbc, hstmt1, rc);
        SQLFreeStmt(hstmt1, SQL_CLOSE);
    }

    printf("Commiting Transaction\n");
    rc = SQLTransact(henv, hdbc, SQL_COMMIT);
    if (rc != SQL_NO_DATA_FOUND )
        check_error (henv, hdbc, SQL_NULL_HSTMT, rc);

    terminate(henv, hdbc);
    return (0);
}/* end main */
```

References

- “SQLExecute - Execute a Statement” on page 45
- “SQLExecDirect - Execute a Statement Directly” on page 41

SQLGetInfo - Get General Information

Purpose

SQLGetInfo() returns general information, (including supported data conversions) about the DBMS that the application is currently connected to.

Syntax

```
SQLRETURN SQLGetInfo (SQLHDBC        hdbc,
                      SQLSMALLINT    fInfoType,
                      SQLPOINTER     rgbInfoValue,
                      SQLSMALLINT    cbInfoValueMax,
                      SQLSMALLINT    *pcbInfoValue);
```

Function Arguments

Table 33. SQLGetInfo Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Database connection handle
SQLSMALLINT	*fInfoType*	input	The type of information desired.
SQLPOINTER	*rgbInfoValue*	output (also input)	Pointer to buffer where this function will store the desired information. Depending on the type of information being retrieved, 4 types of information can be returned: • 16 bit integer value • 32 bit integer value • 32 bit binary value • null-terminated character string
SQLSMALLINT	*cbInfoValueMax*	input	Maximum length of the buffer pointed by *rgbInfoValue* pointer.
SQLSMALLINT *	*pcbInfoValue*	output	Pointer to location where this function will return the total number of bytes available to return the desired information. In the case of string output, this size does not include the null terminating character. If the value in the location pointed to by *pcbInfoValue* is greater than the size of the *rgbInfoValue* buffer as specified in *cbInfoValueMax*, then the string output information would be truncated to *cbInfoValueMax* - 1 bytes and the function would return with SQL_SUCCESS_WITH_INFO.

Usage

Data conversion is discussed in Table 34 on page 67.

In addition to the *fInfoType* values in the table above, specifying the values in the table below will indicate whether the data type can be converted to other data types.

When SQLGetInfo() is called with one of the conversion *fInfoTypes* from the left column, a 32 bit mask is returned. This mask can then be compared (using a logical *and*) with **any** of the values from the right column.

Notice that the symbolic names in both columns are identical except that CONVERT has been replaced with CVT.

Table 34. Data Conversion Values for SQLGetInfo()

fInfoType	**Comparison Mask**
SQL_CONVERT_CHAR [56]	SQL_CVT_CHAR
SQL_CONVERT_NUMERIC [63]	SQL_CVT_NUMERIC
SQL_CONVERT_DECIMAL [58]	SQL_CVT_DECIMAL
SQL_CONVERT_INTEGER [61]	SQL_CVT_INTEGER
SQL_CONVERT_SMALLINT [65]	SQL_CVT_SMALLINT
SQL_CONVERT_FLOAT [60]	SQL_CVT_FLOAT
SQL_CONVERT_REAL [64]	SQL_CVT_REAL
SQL_CONVERT_DOUBLE [59]	SQL_CVT_DOUBLE
SQL_CONVERT_VARCHAR [70]	SQL_CVT_VARCHAR
SQL_CONVERT_LONGVARCHAR [62]	SQL_CVT_LONGVARCHAR
SQL_CONVERT_BINARY [54]	SQL_CVT_BINARY
SQL_CONVERT_VARBINARY [69]	SQL_CVT_VARBINARY
SQL_CONVERT_BIT [55]	SQL_CVT_BIT
SQL_CONVERT_TINYINT [68]	SQL_CVT_TINYINT
SQL_CONVERT_BIGINT [53]	SQL_CVT_BIGINT
SQL_CONVERT_DATE [57]	SQL_CVT_DATE
SQL_CONVERT_TIME [66]	SQL_CVT_TIME
SQL_CONVERT_TIMESTAMP [67]	SQL_CVT_TIMESTAMP
SQL_CONVERT_LONGVARBINARY [71]	SQL_CVT_LONGVARBINARY

Note: Since graphic data types are not supported by ODBC, they are not supported by SQLGetInfo().

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 35 (Page 1 of 2). SQLGetInfo SQLSTATEs

SQLSTATE	**Description**	**Explanation**
01004	Data truncated.	The requested information was returned as a null-terminated string and its length exceeded the length of the application buffer as specified in *cbInfoValueMax*. The argument *pcbInfoValue* contains the actual (not truncated) length of the requested information. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open.	The type of information requested in *fInfoType* requires an open connection. Only SQL_ODBC_VER does not require an open connection.

Table 35 (Page 2 of 2). SQLGetInfo SQLSTATEs

SQLSTATE	Description	Explanation
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	The value specified for argument *cbInfoValueMax* was less than 0. An invalid *fInfoType* was specified. The argument *rgbInfoValue* was a null pointer. The *fInfoType* was SQL_DRIVER_HSTMT and the value pointed to by *rgbInfoValue* was not a valid handle.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.
S1103	Direction option out of range.	
S1C00	Driver not capable.	The value specified in the argument *fInfoType* is not supported by either the driver or the data source.

Restrictions

The ODBC SQLSTATE S1090 will not be returned. Instead, the X/Open S1009 SQLSTATE will be returned.

Example

```
/*******************************************************
** file = getinfo.c
**    - Connect to a database and display database and driver information.
**    - error handling has been ignored for simplicity.
**
**  Functions used:
**
**    SQLAllocConnect  SQLDisconnect
**    SQLAllocEnv      SQLFreeConnect
**    SQLRConnect      SQLFreeEnv
**
**    SQLGetFunctions  SQLGetInfo
*******************************************************/
#include <stdio.h>
#include "sqlcli1.h"
int initialize(SQLHENV *henv,
               SQLHDBC *hdbc);
int terminate(SQLHENV henv,
              SQLHDBC hdbc);
int main()
{
SQLHENV     henv;
SQLHDBC     hdbc;
SQLRETURN   rc;
SQLCHAR     buffer[255];
SQLSMALLINT output;
SQLSMALLINT outlen,
            supported;
    initialize(&henv, &hdbc);
    /* Check to see if SQLGetInfo() is supported */
    rc = SQLGetFunctions(hdbc, SQL_API_SQLGETINFO, &supported);
    if (supported)
    { /* get information about current connection */
        rc = SQLGetInfo(hdbc, SQL_DATA_SOURCE_NAME, buffer, 255, &   outlen);
        printf("   Server Name: %s\n", buffer);
        rc = SQLGetInfo(hdbc, SQL_DATABASE_NAME, buffer, 255, &out   len);
        printf(" Database Name: %s\n", buffer);
        rc = SQLGetInfo(hdbc, SQL_SERVER_NAME, buffer, 255, &outle   n);
        printf(" Instance Name: %s\n", buffer);
        rc = SQLGetInfo(hdbc, SQL_DBMS_NAME, buffer, 255, &outlen)   ;
        printf("     DBMS Name: %s\n", buffer);
        rc = SQLGetInfo(hdbc, SQL_DBMS_VER, buffer, 255, &outlen);
        printf("  DBMS Version: %s\n", buffer);
        rc = SQLGetInfo(hdbc, SQL_DRIVER_NAME, buffer, 255, &outle   n);
        printf("   Driver Name: %s\n", buffer);
        rc = SQLGetInfo(hdbc, SQL_DRIVER_VER, buffer, 255, &outlen   );
        printf("Driver Version: %s\n", buffer);
        rc = SQLGetInfo(hdbc, SQL_ODBC_API_CONFORMANCE, &output,
                    sizeof(output), &outlen);
        switch (output){
         case 0 : strcpy(buffer, "CORE");
                 break;
         case 1 : strcpy(buffer, "Level 1");
                 break;
         case 2 : strcpy(buffer, "Level 2");
                 break;
         default: printf("Error calling getinfo!");
                 return(SQL_ERROR);
        }
        printf("ODBC API Conformance Level: %s\n", buffer);
        rc = SQLGetInfo(hdbc, SQL_ODBC_SQL_CONFORMANCE, &output,
                    sizeof(output), &outlen);
        switch (output){
         case 0 : strcpy(buffer, "Minimum Grammar");
                 break;
         case 1 : strcpy(buffer, "Core Grammar");
                 break;
         case 2 : strcpy(buffer, "Extended Grammar");
                 break;
          default: printf("Error calling getinfo!");
```

```
                 return(SQL_ERROR);
        }
        printf("ODBC SQL Conformance Level: %s\n", buffer);
    }
    else
    {   printf("SQLGetInfo is not supported!\n");
    }
    /*********   Start Processing Step  *************************/
    /* allocate statement handle, execute statement, etc.       */
    /*********   End Processing Step  ***************************/
    terminate(henv, hdbc);
    return (SQL_SUCCESS);
}
```

References

None.

SQLNumResultCols - Get Number of Result Columns

Purpose

SQLNumResultCols() returns the number of columns in the result set associated with the input statement handle.

DataJoiner calls SQLPrepare() or SQLExecDirect() before calling this function.

After calling this function, DataJoinercan call SQLDescribeCol(), SQLBindCol() or SQLGetData().

Syntax

```
SQLRETURN SQLNumResultCols (SQLHSTMT       hstmt,
                            SQLSMALLINT    *pccol);
```

Function Arguments

Table 36. SQLNumResultCols Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLSMALLINT *	*pccol*	output	Number of columns in the result set

Usage

The function sets the output argument to zero if the last statement executed on the input statement handle is not a SELECT.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 37 (Page 1 of 2). SQLNumResultCols SQLSTATEs

SQLSTATE	Description	Explanation
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.

Table 37 (Page 2 of 2). SQLNumResultCols SQLSTATEs

SQLSTATE	Description	Explanation
S1009 (optional)	Invalid argument value.	*pcbCol* was a null pointer.
S1010 (optional)	Function sequence error.	The function was called prior to calling SQLPrepare or SQLExecDirect for the *hstmt*.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to "Example" on page 80.

References

- "SQLBindCol - Bind a Column to DataJoiner Storage" on page 24
- "SQLDescribeCol - Describe Column Attributes" on page 30
- "SQLExecDirect - Execute a Statement Directly" on page 41
- "SQLPrepare - Prepare a Statement" on page 73

SQLPrepare - Prepare a Statement

Purpose

SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the DBMS to be prepared. DataJoiner references this prepared statement by passing the statement handle to other functions.

If the statement handle has been previously used with a SELECT statement, DataJoiner calls SQLFreeStmt() to close the cursor, before calling SQLPrepare().

Syntax

```
SQLRETURN SQLPrepare (SQLHSTMT       hstmt,
                      SQLCHAR        *szSqlStr,
                      SQLINTEGER     cbSqlStr);
```

Function Arguments

Table 38. SQLPrepare Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle. There must not be an open cursor associated with *hstmt*.
SQLCHAR *	*szSqlStr*	input	SQL statement string
SQLINTEGER	*cbSqlStr*	input	Length of contents of *szSqlStr* argument. This will be set to either the exact length of the SQL statement in *szSqlstr*, or to SQL_NTS if the statement text is null-terminated.

Usage

Once a statement has been prepared using SQLPrepare(), DataJoiner can request information about the format of the result set (if it was a SELECT statement) by calling:

- SQLNumResultCols()
- SQLDescribeCol()

A prepared statement can be executed once, or multiple times by calling SQLExecute(). The SQL statement remains associated with the statement handle until the handle is used with another SQLPrepare() call or SQLExecDirect().

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 the value DataJoiner storage is to be substituted, when SQLExecute() is called. SQLSetParam() is used to bind (or associate) DataJoiner storage to each parameter marker, and to indicate if any data conversion should be performed at the time that the data is transferred.

The SQL statement cannot be a COMMIT or ROLLBACK. `SQLTransact()` must be called to issue COMMIT or ROLLBACK.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement will be defined on a separate statement handle under the same connection handle.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 39 (Page 1 of 2). SQLPrepare SQLSTATEs

SQLSTATE	Description	Explanation
01504 *****	No WHERE clause.	*szSqlStr* contained an UPDATE or DELETE statement which did not contain a WHERE clause.
01508 *****	No blocking.	The statement was disqualified for blocking for reasons other than storage.
21S01	Insert value list does not match column list.	*szSqlStr* contained an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*szSqlStr* contained a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
24000 (optional)	Invalid cursor state.	There was an open cursor on the specified *hstmt*.
34000	Invalid cursor name.	*szSqlStr* contained a Positioned DELETE or a Positioned UDPATE and the cursor referenced by the statement being executed was not open.
37xxx	Syntax error or access violation.	*szSqlStr* contained one or more of the following: • a COMMIT • a ROLLBACK • an SQL statement that the connected database server could not prepare • a statement containing a syntax error
40000 *****	Serialization failure.	The transaction to which this SQL statement belonged was rolled back due to deadlock or timeout.
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
42xxx	Syntax error or access violation.	The current user did not have permission to execute the SQL statement in *szSqlstr*.

Table 39 (Page 2 of 2). SQLPrepare SQLSTATEs

SQLSTATE	Description	Explanation
58004	System error.	Unrecoverable system error.
S0001	Base table or view already exists.	*szSqlStr* contained a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already existed.
S0002	Table or view not found.	*szSqlStr* contained an SQL statement that references a table name or a view name which did not exist.
S0011	Index already exists.	*szSqlStr* contained a CREATE INDEX statement and the specified index name already existed.
S0012	Index not found.	*szSqlStr* contained a DROP INDEX statement and the specified index name did not exist.
S0021	Column already exists.	*szSqlStr* contained an ALTER TABLE statement and the column specified in the ADD clause was not unique or identified an existing column in the base table.
S0022	Column not found.	*szSqlStr* contained an SQL statement that references a column name which did not exist.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1009 (optional)	Invalid argument value.	*szSqlStr* was a null pointer. The argument *cbSqlStr* was less than 1, but not equal to SQL_NTS.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.

Note: Not all DBMSs report all of the above diagnostic messages at prepare time. Therefore DataJoiner also handles these conditions when calling `SQLExecute()`.

Restrictions

None.

Example

```
/**************************************************************************
** file = prepare.c
**
** Example of preparing then repeatedly executing an SQL statement.
**
** Functions used:
**
**        SQLAllocConnect      SQLFreeConnect
**        SQLAllocEnv          SQLFreeEnv
**        SQLAllocStmt         SQLFreeStmt
**        SQLConnect           SQLDisconnect
**
**        SQLBindCol           SQLFetch
**        SQLTransact          SQLError
**        SQLPrepare           SQLSetParam
**        SQLExecute
**************************************************************************/

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

#define MAX_STMT_LEN 255

int initialize(SQLHENV *henv,
               SQLHDBC *hdbc);

int terminate(SQLHENV henv,
              SQLHDBC hdbc);

int print_error (SQLHENV    henv,
                 SQLHDBC    hdbc,
                 SQLHSTMT   hstmt);

int check_error (SQLHENV    henv,
                 SQLHDBC    hdbc,
                 SQLHSTMT   hstmt,
                 SQLRETURN  rc);

/*******************************************************************
** main
** - initialize
** - terminate
*******************************************************************/
int main()
{
    SQLHENV     henv;
    SQLHDBC     hdbc;
    SQLCHAR     sqlstmt[MAX_STMT_LEN + 1]="";
    SQLRETURN   rc;

    rc = initialize(&henv, &hdbc);
    if (rc == SQL_ERROR) return(terminate(henv, hdbc));

    {SQLHSTMT   hstmt;
     SQLCHAR    sqlstmt[]="SELECT deptname, location from org where division = ?";
     SQLCHAR    deptname[15],
                location[14],
                division[11];

     SQLINTEGER rlength,
                plength;

        rc = SQLAllocStmt(hdbc, &hstmt);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, SQL_NULL_HSTMT, rc);
        /* prepare statement for multiple use */
        rc = SQLPrepare(hstmt, sqlstmt, SQL_NTS);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt, rc);

        /* bind division to parameter marker in sqlstmt */
        rc = SQLSetParam(hstmt, 1, SQL_C_CHAR, SQL_CHAR, 10, 10, division,
                     &plength);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt, rc);

        /* bind deptname to first column in the result set */
        rc = SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) deptname, 15,
                        &rlength);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt, rc);
        rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) location, 14,
                         &rlength);
        if (rc != SQL_SUCCESS )
            check_error (henv, hdbc, hstmt, rc);

        printf("\nEnter Division Name or 'q' to quit:\n");
        printf("(Eastern, Western, Midwest, Corporate)\n");
        gets(division);
        plength = SQL_NTS;

        while(division[0] != 'q')
        {
            rc = SQLExecute(hstmt);
            if (rc != SQL_SUCCESS )
                check_error (henv, hdbc, hstmt, rc);

            printf("Departments in %s Division:\n", division);
            printf("DEPTNAME       Location\n");
            printf("-------------- -------------\n");

            while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
            {
                printf("%-14.14s %-13.13s \n", deptname, location);
            }
            if (rc != SQL_NO_DATA_FOUND )
                check_error (henv, hdbc, hstmt, rc);
            SQLFreeStmt(hstmt, SQL_CLOSE);
            printf("\nEnter Division Name or 'q' to quit:\n");
            printf("(Eastern, Western, Midwest, Corporate)\n");
            gets(division);
        }
    }

    rc = SQLTransact(henv, hdbc, SQL_ROLLBACK);
    if (rc != SQL_SUCCESS )
```

```
        check_error (henv, hdbc, SQL_NULL_HSTMT, rc);

    terminate(henv, hdbc);
    return (0);
}/* end main */
```

References

- “SQLDescribeCol - Describe Column Attributes” on page 30
- “SQLExecDirect - Execute a Statement Directly” on page 41
- “SQLExecute - Execute a Statement” on page 45
- “SQLNumResultCols - Get Number of Result Columns” on page 71

SQLRowCount - Get Row Count

Purpose

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.

DataJoiner calls SQLExecute() or SQLExecDirect() before calling this function.

Syntax

```
SQLRETURN SQLRowCount (SQLHSTMT      hstmt,
                       SQLINTEGER    *pcrow);
```

Function Arguments

Table 40. SQLRowCount Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLINTEGER *	*pcrow*	output	Pointer to location where the number of rows affected is stored.

Usage

If the last executed statement referenced by the input statement handle 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.

Any rows in other tables that can have been affected by the statement (for example, cascading deletes) are not included in the count.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 41 (Page 1 of 2). SQLRowCount SQLSTATEs

SQLSTATE	Description	Explanation
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.

Table 41 (Page 2 of 2). SQLRowCount SQLSTATEs

SQLSTATE	Description	Explanation
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1009 (optional)	Invalid argument value.	*pcrow* was a null pointer.
S1010 (optional)	Function sequence error.	The function was called prior to calling SQLExecute or SQLExecDirect for the *hstmt*.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

```
/*************************************************************************
** file = typical.c
**
** 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;


    SQLAllocStmt (hdbc, &hstmt);       /* allocate a statement handle */

    /* 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_error (henv, hdbc, hstmt, rc);

    rc = SQLNumResultCols (hstmt, &nresultcols);
    if (rc != SQL_SUCCESS)
      check_error (henv, hdbc, hstmt, rc);

    /* determine statement type */
    if (nresultcols == 0) /* statement is not a select statement */
    {
        rc = SQLRowCount (hstmt, &rowcount);
        if (rowcount > 0 ) /* assume statement is UPDATE, INSERT, DELETE */
        {
            printf ("Statement executed, %ld rows affected\n", rowcount);
        }
      else  /* assume statement is GRANT, REVOKE or a DLL statement */
        {
              printf ("Statement completed successful\n");
        }
    }
    else /* display the result set */
    {
        display_results(hstmt, nresultcols);
    } /* end determine statement type */

    SQLFreeStmt (hstmt, SQL_DROP );       /* free statement handle */

    return (0);
}/* end process_stmt */
```

References

- “SQLExecDirect - Execute a Statement Directly” on page 41
- “SQLExecute - Execute a Statement” on page 45
- “SQLNumResultCols - Get Number of Result Columns” on page 71

SQLSetParam - Set Parameter

Purpose

`SQLSetParam()` associates (binds) DataJoiner storage to a parameter marker in an SQL statement. When the statement is later executed, the contents of the bound variables are sent to the database server. This function is also used to specify any required data conversion.

Syntax

```
SQLRETURN SQLSetParam (SQLHSTMT       hstmt,
                       SQLSMALLINT    ipar,
                       SQLSMALLINT    fCType,
                       SQLSMALLINT    fSqlType,
                       SQLINTEGER     cbParamDef,
                       SQLSMALLINT    ibScale,
                       SQLPOINTER     rgbValue,
                       SQLINTEGER     *pcbValue);
```

Function Arguments

Table 42 (Page 1 of 3). SQLSetParam Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLSMALLINT	*ipar*	input	Parameter marker number
SQLSMALLINT	*fCType*	input	C data type of argument. The following types are supported: • SQL_C_CHAR • SQL_C_DATE • SQL_C_DEFAULT • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TIME • SQL_C_TIMESTAMP Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type for the type indicated in *fSqlType*. SQL_C_DEFAULT can not be specified for DECIMAL, NUMERIC or any of the graphic data types.

Table 42 (Page 2 of 3). SQLSetParam Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	*fSqlType*	input	SQL Data Type of column. The supported types are: • SQL_CHAR • SQL_DATE • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC (SQL/400 only) • SQL_REAL • SQL_SMALLINT • SQL_TIME • SQL_TIMESTAMP • SQL_VARCHAR • SQL_VARGRAPHIC If the SQL data type for the column is SQL_GRAPHIC, SQL_VARGRAPHIC, or SQL_LONGVARGRAPHIC then *fCType* is not allowed to be anything else except SQL_C_CHAR; otherwise, an error will occur during the execution of the statement (and an SQLSTATE of **07**006 will be generated).
SQLINTEGER	*cbParamDef*	input	Precision of the corresponding parameter marker. If *fSqlType* is: • SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR, this is the maximum length of the argument • SQL_GRAPHIC, SQL_VARGRAPHIC, SQL_LONGVARGRAPHIC, this is the maximum number of double-byte characters for this argument • SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision; otherwise, this argument is unused
SQLSMALLINT	*ibScale*	input	Scale of the corresponding parameter marker if *fSqlType* is SQL_DECIMAL or SQL_NUMERIC. If *fSqlType* is SQL_TIMESTAMP, this is the number of digits to the right of the decimal point in the character representation of a timestamp (for example, the scale of yyyy-mm-dd hh:mm:ss.fff is 3). Other than for the *fSqlType* values mentioned here, *ibScale* is unused.
SQLPOINTER	*rgbValue*	input (deferred)	Pointer to the location which contains (when the statement is executed) the actual values for the associated parameter marker.

Table 42 (Page 3 of 3). SQLSetParam Arguments

Data Type	Argument	Use	Description
SQLINTEGER *	*pcbValue*	input (deferred)	Pointer to the location which contains (when the statement is executed) the length of the parameter marker value stored at *rgbValue*. To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA. If *fCType* is SQL_C_CHAR, this storage location must contain either the exact length of the data stored at *rgbValue*, or SQL_NTS if the contents at *rgbValue* is null-terminated. If *fCType* indicates character data (explicitly, or implicitly using SQL_C_DEFAULT), and this pointer is set to NULL, DataJoiner will always provide a null-terminated string in *rgbValue*. This also implies that this parameter marker will never have a null value. If *fSqlType* denotes a graphic data type, the pointer to *pcbValue* can never be NULL and the contents of *pcbValue* can never hold SQL_NTS. This length should be the number of octets that the double byte data occupies; therefore, the length should always be a multiple of 2. In fact, if the length is odd, then an error will occur when the statement is executed.

Usage

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

The pointers, *rgbValue* and *pcbValue*, remain valid until the application has finished executing the statement. The data in *rgbValue* must be in the form specified by the *fCType* argument.

DataJoiner binds a variable to each parameter marker in the SQL statement before executing the SQL statement. `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.

All parameters bound by this function remain in effect until `SQLFreeStmt()` is called with the SQL_DROP option, or until `SQLSetParam()` is called again for the same parameter *ipar* number.

After the SQL statement has been executed, and the results processed, DataJoiner can reuse the statement handle to execute a different SQL statement.

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

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 `SQLDirectExec()` is called.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 43 (Page 1 of 2). SQLSetParam SQLSTATEs

SQLSTATE	Description	Explanation
07006	Restricted data type attribute violation.	The data value identified by the *fCType* argument cannot be converted to the data type identified by the *fSqlType* argument. The argument *fCType* is SQL_C_DEFAULT and the argument *fSqlType* is one of SQL_DECIMAL, SQL_NUMERIC, SQL_GRAPHIC, SQL_VARGRAPHIC, or SQL_LONGVARGRAPHIC.
40003 *****	Statement completion unknown.	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1003	Program type out of range.	The value specified by the argument *fCType* not a valid data type or SQL_C_DEFAULT.
S1004	SQL data type out of range.	The value specified for the argument *fSqlType* is not a valid SQL data type.
S1009 (optional)	Invalid argument value.	The argument *rgbValue* was a null pointer and the argument *pcbValue* was a null pointer. The value specified for the argument *ipar* was less than 1 or greater than the maximum number of parameters supported by the data source. The value for the argument *fSqlType* was either SQL_DECIMAL or SQL_NUMERIC and the value for the argument *cbParamDef* was less than 1. The value for the argument *fSqlType* is either SQL_DECIMAL or SQL_NUMERIC and the value for the argument *ibScale* was less than 0 or greater than the value for the argument *cbParamDef*. The value for the argument *fCType* was SQL_C_TIMESTAMP and the value for the argument *fSqlType* was either SQL_CHAR or SQL_VARCHAR and the value for the argument *ibScale* was less than 0 or greater than 6.

Table 43 (Page 2 of 2). SQLSetParam SQLSTATEs

SQLSTATE	Description	Explanation
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.
S1C00	Driver not capable.	The driver or data source does not support the conversion specified by the combination of the value specified for the argument *fCType* and the value specified for the argument *fSqlType*. The value specified for the argument *fCType* or *fSqlType* is not supported by either the driver or the data source.

Restrictions

None.

Example

Refer to “Example” on page 75.

References

- “SQLExecDirect - Execute a Statement Directly” on page 41
- “SQLExecute - Execute a Statement” on page 45
- “SQLPrepare - Prepare a Statement” on page 73

SQLTransact - Transaction Management

Purpose

SQLTransact() commits or rolls back the current transaction in the connection.

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, DataJoinercalls SQLTransact() before it disconnects from the database.

Syntax

```
SQLRETURN SQLTransact (SQLHENV        henv,
                       SQLHDBC        hdbc,
                       SQLSMALLINT    fType);
```

Function Arguments

Table 44. SQLTransact Arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle. If *hdbc* is a valid connection handle, *henv* is ignored.
SQLHDBC	*hdbc*	input	Database connection handle. If *hdbc* is set to SQL_NULL_HDBC, then *henv* will contain the environment handle that the connection is associated with.
SQLSMALLINT	*fType*	input	The desired action for the transaction. The value for this argument must be one of: • SQL_COMMIT • SQL_ROLLBACK

Usage

In the generic access API, a transaction begins implicitly when a DataJoiner instance that does not already have an active transaction issues SQLPrepare(), or SQLExecDirect(). The transaction ends when DataJoinercalls SQLTransact().

Completing a transaction has the following effects:

- Prepared SQL statements do not survive transactions. DataJoinerprepares statements again in order to execute them as part of a new transaction. This means that statement handles are still valid after a call to SQLTransact(), and can be reused for subsequent SQL statements or deallocated by calling SQLFreeStmt().
- Cursor names, bound parameters, and column bindings survive transactions.

- Open cursors are closed and any result sets that are pending retrieval are discarded.

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, DataJoiner can be unable to determine whether the COMMIT or ROLLBACK has been processed, and a database administrator's help can be required. Refer to the DBMS product information for more information on transaction logs and other transaction management tasks.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 45. SQLTransact SQLSTATEs

SQLSTATE	Description	Explanation
08003 (optional)	Connection not open.	The *hdbc* was not in a connected state.
08007	Connection failure during transaction.	The connection associated with the *hdbc* failed during the execution of the function during the execution of the function and it cannot be determined whether the requested COMMIT or ROLLBACK occurred before the failure.
58004	System error.	Unrecoverable system error.
S1001	Memory allocation failure.	The driver is unable to allocate memory required to support execution or completion of the function.
S1012 (optional)	Invalid transaction operation state.	The value specified for the argument *fType* was neither SQL_COMMIT not SQL_ROLLBACK.
S1013 *****	Memory management problem.	The driver was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 49.

Appendix A. General Diagnostic Information

This appendix contains information that various sections in this book refer to.

Return Codes

Table 46. The Generic Access API Function Return Codes.

Return Code	Value	Description
SQL_SUCCESS	0	The function completed successfully; no additional SQLSTATE information is available.
SQL_SUCCESS_WITH_INFO	1	The function completed successfully, with a warning or other information. Call `SQLError()` to receive the SQLSTATE and other error information.
SQL_NO_DATA_FOUND	100	The function returned successfully, but no relevant information was found.
SQL_ERROR	-1	The function failed. Call `SQLError()` to receive the SQLSTATE and any other error information.
SQL_INVALID_HANDLE	-2	The function failed due to an invalid handle (environment, connection or statement handle) passed as an input argument.

SQLSTATE Cross Reference

Table 47 cross-references all the SQLSTATEs that are listed in the *Diagnostics* section of each function description in Chapter 3, “Required Functions” on page 13.

Table 47 (Page 1 of 6). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
01002	Disconnect error.	• `SQLDisconnect()`
01004	Data truncated.	• `SQLDescribeCol()` • `SQLFetch()` • `SQLGetCursorName()`
01504 *	No WHERE clause.	• `SQLExecDirect()` • `SQLExecute()` • `SQLPrepare()`
01508 *	No blocking.	• `SQLExecDirect()` • `SQLExecute()` • `SQLPrepare()`
07001	Wrong number of parameters.	• `SQLExecDirect()` • `SQLExecute()`
07002 *	Invalid column number.	• `SQLFetch()`

Table 47 (Page 2 of 6). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
07005 *	Not a SELECT statement.	• SQLDescribeCol()
07006	Restricted data type attribute violation.	• SQLExecDirect() • SQLExecute() • SQLFetch() • SQLSetParam()
08001	Unable to connect to data source.	• SQLConnect()
08002	Connection is use.	• SQLConnect()
08003	Connection not open.	• SQLAllocStmt() • SQLDisconnect() • SQLTransact()
08004	Data source rejected establishment of connection.	• SQLConnect()
08007	Connection failure during transaction.	• SQLTransact()
21S01	Insert value list does not match column list.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
21S02	Degrees of derived table does not match column list.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
22001	String data right truncation.	• SQLExecDirect() • SQLExecute()
22002	Invalid length buffer.	• SQLFetch()
22003	Numeric value out of range.	• SQLExecDirect() • SQLExecute() • SQLFetch()
22005	Error in assignment.	• SQLExecDirect() • SQLExecute() • SQLFetch()
22007 *	Invalid date time format.	• SQLExecDirect() • SQLExecute() • SQLFetch()
22008	Datetime field overflow.	• SQLExecDirect() • SQLExecute() • SQLFetch()
22012	Division by zero.	• SQLExecDirect() • SQLExecute() • SQLFetch()
23000	Integrity constraint violation.	• SQLExecDirect() • SQLExecute()
24000	Invalid cursor state.	• SQLExecDirect() • SQLExecute() • SQLFetch() • SQLPrepare()

Table 47 (Page 3 of 6). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
24504 *	Invalid cursor state.	• SQLExecDirect() • SQLExecute()
25000	Invalid transaction state.	• SQLDisconnect()
28000	Invalid authorization specification.	• SQLConnect()
34000	Invalid cursor name.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
37xxx	Syntax error or access violation.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
40000 *	Serialization failure.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
40003 *	Statement completion unknown.	• SQLAllocStmt() • SQLBindCol() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLFetch() • SQLFreeStmt() • SQLGetCursorName() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetParam()
42xxx	Syntax error or access violation.	• SQLPrepare()
42504	Syntax error or access violation.	• SQLExecDirect() • SQLExecute()
44000	Integrity constraint violation.	• SQLExecDirect() • SQLExecute()

Table 47 (Page 4 of 6). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
58004	System error.	• SQLAllocEnv() • SQLAllocStmt() • SQLBindCol() • SQLConnect() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLFetch() • SQLFreeConnect() • SQLFreeEnv() • SQLFreeStmt() • SQLGetCursorName() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetParam() • SQLTransact()
S0001	Base table or view already exists.	• SQLPrepare()
S0002	Table or view not found.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
S0011	Index already exists.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
S0012	Index not found.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
S0021	Column already exists.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
S0022	Column not found.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
S1000	General error.	•

Table 47 (Page 5 of 6). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
S1001	Memory allocation failure.	• SQLAllocConnect() • SQLAllocStmt() • SQLBindCol() • SQLConnect() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLFetch() • SQLFreeConnect() • SQLFreeEnv() • SQLFreeStmt() • SQLGetCursorName() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetParam() • SQLTransact()
S1002	Invalid column number.	• SQLBindCol() • SQLDescribeCol()
S1003	Program type out of range.	• SQLBindCol() • SQLSetParam()
S1004	SQL data type out of range.	• SQLSetParam()
S1009	Invalid argument value.	• SQLAllocConnect() • SQLAllocStmt() • SQLBindCol() • SQLConnect() • SQLDescribeCol() • SQLExecDirect() • SQLFreeStmt() • SQLGetCursorName() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetParam()
S1010	Function sequence error.	• SQLDescribeCol() • SQLExecute() • SQLFetch() • SQLFreeConnect() • SQLFreeEnv() • SQLGetCursorName() • SQLNumResultCols() • SQLRowCount()
S1012	Invalid transaction operation state.	• SQLTransact()

Table 47 (Page 6 of 6). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
S1013 *	Memory management problem.	• SQLAllocConnect() • SQLAllocStmt() • SQLBindCol() • SQLConnect() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLFetch() • SQLFreeConnect() • SQLFreeEnv() • SQLGetCursorName() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetParam() • SQLTransact()
S1014 *	Out of handles.	• SQLAllocConnect() • SQLAllocStmt()
S1015	No cursor name available.	• SQLGetCursorName()
S1103	Direction option out of range.	•
S1501 *	Invalid data source name.	• SQLConnect()
S1C00	Driver not capable.	• SQLBindCol() • SQLDescribeCol() • SQLFetch() • SQLSetParam()

Appendix B. Data Conversion

This section contains tables that are used for data conversion between C and SQL data types. This includes:

- SQL and C data types
- Precision, scale, length, and display size of each data type
- Supported data conversion
- Conversion from SQL to C data types
- Conversion from C to SQL data types

Data Types

This section is intended to be used as a reference.

Table 48 (Page 1 of 2). SQL Data Types and Default C Data Types

SQL Type SQL Symbolic	C Symbolic	Generic C Type ODBC C Type	Actual C type
CHAR SQL_CHAR	SQL_C_CHAR	SQLCHAR UCHAR	unsigned char
VARCHAR SQL_VARCHAR	SQL_C_CHAR	SQLCHAR UCHAR	unsigned char
LONGVARCHAR SQL_LONGVARCHAR	SQL_C_CHAR	SQLCHAR UCHAR	unsigned char
GRAPHIC 1 SQL_GRAPHIC	SQL_C_CHAR	SQLCHAR UCHAR	unsigned char
VARGRAPHIC [1] SQL_VARGRAPHIC	SQL_C_CHAR	SQLCHAR UCHAR	unsigned char
LONGVARGRAPHIC[1] SQL_LONGVARGRAPHIC	SQL_C_CHAR	SQLCHAR UCHAR	unsigned char
SMALLINT SQL_SMALLINT	SQL_C_SHORT	SQLSMALLINT SWORD	short int
INTEGER SQL_INTEGER	SQL_C_LONG	SQLINTEGER SDWORD	long int
DECIMAL [1] SQL_DECIMAL	SQL_C_CHAR	SQLCHAR UCHAR	unsigned char
NUMERIC [1] SQL_NUMERIC	SQL_C_CHAR	SQLCHAR UCHAR	unsigned char
DOUBLE SQL_DOUBLE	SQL_C_DOUBLE	SQLDOUBLE SDOUBLE	double
FLOAT SQL_FLOAT	SQL_C_DOUBLE	SQLDOUBLE SDOUBLE	double
REAL SQL_REAL	SQL_C_FLOAT	SQLREAL SFLOAT	float
DATE SQL_DATE	SQL_C_DATE	DATE_STRUCT	see Table 49 on page 96

Table 48 (Page 2 of 2). SQL Data Types and Default C Data Types

SQL Type SQL Symbolic	C Symbolic	Generic C Type ODBC C Type	Actual C type
TIME SQL_TIME	SQL_C_TIME	TIME_STRUCT	See Table 49 on page 96.
TIMESTAMP SQL_TIMESTAMP	SQL_C_TIMESTAMP	TIMESTAMP_STRUCT	See Table 49 on page 96.

Note: 1 SQL_C_DEFAULT cannot be used for SQL_DECIMAL, SQL_NUMERIC, SQL_GRAPHIC, SQL_VARGRAPHIC, and SQL_LONGVARGRAPHIC. For these data types you must specify the C symbolic data type (SQL_C_CHAR) explicitly.

Table 49. C DATE, TIME, and TIMESTAMP Structures

C Type	Generic Structure	ODBC Structure

DATE_STRUCT	typedef struct DATE_STRUCT
	{
	SQLSMALLINT year;
	SQLSMALLINT month;
	SQLSMALLINT day;
	} DATE_STRUCT;
		typedef struct tagDATE_STRUCT
		{
		SWORD year;
		UWORD month;
		UWORD day;
		} DATE_STRUCT;

TIME_STRUCT	typedef struct TIME_STRUCT
	{
	SQLSMALLINT hour;
	SQLSMALLINT minutes;
	SQLSMALLINT second;
	} TIME_STRUCT;
		typedef struct tagTIME_STRUCT
		{
		UWORD hour;
		UWORD minutes;
		UWORD second;
		} TIME_STRUCT;

TIMESTAMP_STRUCT	typedef struct TIMESTAMP_STRUCT
	{
	SQLSMALLINT year;
	SQLSMALLINT month;
	SQLSMALLINT day;
	SQLSMALLINT hour;
	SQLSMALLINT minute;
	SQLSMALLINT second;
	SQLINTEGER fraction;
	} TIMESTAMP_STRUCT;
		typedef struct tagTIMESTAMP_STRUCT
		{
		SWORD year;
		UWORD month;
		UWORD day;
		UWORD hour;
		UWORD minute;
		UWORD second;
		UDWORD fraction;
		} TIMESTAMP_STRUCT;

Other C Data Types

Table 50. Generic Data Types and Actual C Data Types (DOS, UNIX, OS/2)

Symbolic Type	Actual 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 database connection information.
SQLHSTMT	long int	Handle referencing statement information.
SQLRETURN	long int	Return code from the generic access API functions.

Table 51. ODBC Argument Data Types and Actual C Data Types (Windows)

ODBC Type	Actual C Type	Typical Usage
PTR	void FAR *	Pointers to storage for data and parameters.
HENV	void FAR *	Handle referencing environment information.
HDBC	void FAR *	Handle referencing database connection information.
HSTMT	void FAR *	Handle referencing statement information.
RETCODE	int	Return code from the generic access API functions.

Precision

The precision of a numeric column or parameter refers to the maximum number of digits that areused by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter. The following table defines the precision for each SQL data type.

Table 52. Precision

fSqlType	Precision
SQL_CHAR SQL_VARCHAR	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_DOUBLE [b]	15
SQL_DATE [b]	10 (the number of characters in the yyyy-mm-dd format).
SQL_TIME [b]	8 (the number of characters in the hh:mm:ss format).
SQL_TIMESTAMP	The number of characters in the “yyy-mm-dd hh:mm:ss[.fff[fff]]” format used by the TIMESTAMP data type. For example, if a timestamp does not use seconds or fractional seconds, the precision is 16 (the number of characters in the “yyyy-mm-dd hh:mm” format). If a timestamp uses thousandths of a second, the precision is 23 (the number of characters in the “yyyy-mm-dd hh:mm:ss.fff” format).
SQL_GRAPHIC SQL_VARGRAPHIC	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 **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 **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 53. Scale

fSqlType	Scale
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	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_DATE SQL_TIME	Not applicable.
SQL_TIMESTAMP	The number of digits to the right of the decimal point in the “yyyy-mm-dd hh:mm:ss[fff[fff]]” format. For example, if the TIMESTAMP data type uses the “yyyy-mm-dd hh:mm:ss.fff” format, the scale is 3.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC	Not applicable.

Length

The length of a column is the maximum number of *bytes* returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte. Note that the length of a column may be different than the number of bytes required to store the data on the data source. For a list of default C data types, see the "Default C Data Types” section.

The following table defines the length for each SQL data type.

Data Types

Table 54. Length

fSqlType	**Length**
SQL_CHAR SQL_VARCHAR	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_FLOAT	8 (eight bytes).
SQL_DOUBLE	8 (eight bytes).
SQL_DATE SQL_TIME	6 (the size of the DATE_STRUCT or TIME_STRUCT structure).
SQL_TIMESTAMP	16 (the size of the TIMESTAMP_STRUCT structure).
SQL_GRAPHIC SQL_VARGRAPHIC	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 55. Display Size

fSqlType	Display Size
SQL_CHAR SQL_VARCHAR	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_FLOAT SQL_DOUBLE	22 (a sign, 15 digits, a decimal point, the letter E, a sign, and 3 digits).
SQL_DATE	10 (a date in the format yyyy-mm-dd).
SQL_TIME	8 (a time in the format hh:mm:ss).
SQL_TIMESTAMP	19 (if the scale of the timestamp is 0) or 20 plus the scale of the timestamp (if the scale is greater than 0). This is the number of characters in the "yyyy-mm-dd hh:mm:ss[fff[fff]]" format. For example, the display size of a column storing thousandths of a second is 23 (the number of characters in "yyyy-mm-dd hh:mm:ss.fff").
SQL_GRAPHIC SQL_VARGRAPHIC	The defined length of the column or parameter. For example, the display size of a column defined as CHAR(10) is 10.
SQL_LONGVARGRAPHIC	The maximum length of the column or parameter. [a]

Supported Data Conversion

SQL to C Data Types

Table 56. Supported Data Conversions

Source Data Type	LONGVARGRAPHIC	VARGRAPHIC	GRAPHIC	LONGVARBINARY	TIMESTAMP	TIME	DATE	BIGINT	TINYINT	BIT	VARBINARY	BINARY	LONGVARCHAR	VARCHAR	DOUBLE	REAL	FLOAT	SMALLINT	INTEGER	DECIMAL	NUMERIC	CHAR
CHAR VARCHAR					X	X	X						1	X	X	2		X	X	X	3	X
LONGVARCHAR													1	1								1
GRAPHIC VARGRAPHIC													1	X								X
LONGVAR- GRAPHIC													1	1								1
INTEGER SMALLINT DECIMAL NUMERIC DOUBLE FLOAT														X	X	2		X	X	X	3	X
DATE					X		X							X								X
TIME					X	X								X								X
TIMESTAMP					X	X	X							X								X

Converting Data from SQL to C Data Types

The tables in the following sections describe how the driver of the data source converts data that is retrieved from the data source. 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** or **SQLGetData**.

SQL to C Data Types

The tables list the conversions that are defined by ODBC to be valid for a given SQL data type.

If the *fCType* argument in **SQLBindCol** or **SQLGetData** contains a value not shown in the table for a given SQL data type, **SQLFetch**, or **SQLGetData** returns the SQLSTATE 07006 (Restricted data type attribute violation).

If the *fCType* argument contains a value that is shown in the table but that 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:

Length of data
The total length of the data after it has been converted to the specified C data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is returned to the application.

Significant digits
The minus sign (if needed) and the digits to the left of the decimal point.

Display size
The total number of bytes needed to display data in the character format.

Converting Character SQL Data to C Data

The character SQL data types are:

SQL_CHAR
SQL_VARCHAR
SQL_LONGVARCHAR

SQL to C Data Types

Table 57. Converting Character SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Length of data < cbValueMax	Data	Length of data	00000
	Length of data >= cbValueMax	Truncated data	Length of data	01004
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_DOUBLE	Data converted without truncation[a]	Data	Size of the C data type	00000
	Data converted with truncation, but without loss of significant digits[a]	Data	Size of the C data type	01004
	Conversion of data would result in loss of significant digits[a]	Untouched	Size of the C data type	22003
	Data is not a number[a]	Untouched	Size of the C data type	22005
SQL_C_DATE	Data value is a valid date[a]	Data	6[b]	00000
	Data value is not a valid date[a]	Untouched	6[b]	22007
SQL_C_TIME	Data value is a valid time[a]	Data	6[b]	00000
	Data value is not a valid time[a]	Untouched	6[b]	22008
SQL_C_TIMESTAMP	Data value is a valid timestamp [a]	Data	16 [b]	00000
	Data value is not a valid timestamp [a]	Untouched	16 [b]	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 **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

Converting Graphic SQL Data to C Data

The graphic SQL data types are:

SQL_GRAPHIC
SQL_VARGRAPHIC
SQL_LONGVARGRAPHIC

Table 58. Converting Graphic SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Number of double byte characters * 2 <= cbValueMax	Data	Length of data(octects)	00000
	Number of double byte characters * 2 > cbValueMax	Truncated data, to the nearest even byte that is less than *cbValueMax*.	Length of data(octects)	01004

Note:

SQLSTATE **00**000 is not returned by `SQLError()`; 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

SQL to C Data Types

Table 59. Converting Numeric SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Length of data	00000
	Number of significant digits < cbValueMax	Truncated data	Length of data	01004
	Number of significant digits >= cbValueMax	Untouched	Length of data	22003
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_DOUBLE	Data converted without truncation[a]	Data	Size of the C data type	00000
	Data converted with truncation, but without loss of significant digits[a]	Truncated data	Size of the C data type	01004
	Conversion of data would result in loss of significant digits[a]	Untouched	Size of the C data type	22003

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

SQLSTATE **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

Converting Date SQL Data to C Data

The date SQL data type is:

SQL_DATE

Table 60. Converting Date SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 11	Data	10	00000
	cbValueMax < 11	Untouched	10	22003
SQL_C_DATE	None[a]	Data	6[b]	00000
SQL_C_TIMESTAMP	None[a]	Data[c]	16[b]	00000

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The date fields of the TIMESTAMP_STRUCT structure are set to zero.

SQLSTATE **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

When the date SQL data type is converted to the character C data type, the resulting string is in the “yyyy-mm-dd” format.

Converting Time SQL Data to C Data

The time SQL data type is:

SQL_TIME

Table 61. 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_DATE	None[a]	Data	6[b]	00000
SQL_C_TIMESTAMP	None[a]	Data[c]	16[b]	00000

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The date fields of the TIMESTAMP_STRUCT structure are set to zero.

SQLSTATE **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

When the time SQL data type is converted to the character C data type, the resulting string is in the "hh:mm:ss" format.

Converting Timestamp SQL Data to C Data

The timestamp SQL data type is:

SQL_TIMESTAMP

SQL to C Data Types

Table 62. Converting Timestamp SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Length of data	00000
	19 <= cbValueMax <= Display size	Truncated Data[b]	Length of data	01004
	cbValueMax < 19	Untouched	Length of data	22003
SQL_C_DATE	None[a]	Truncated data[c]	6[e]	01004
SQL_C_TIME	None[a]	Truncated data[d]	6[e]	01004
SQL_C_TIMESTAMP	None[a]	Data	16[e]	00000

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] The fractional seconds of the timestamp are truncated.

[c] The time portion of the timestamp is deleted.

[d] The date portion of the timestamp is deleted.

[e] This is the size of the corresponding C data type.

SQLSTATE **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

When the timestamp SQL data type is converted to the character C data type, the resulting string is in the “yyyy-mm-dd hh:mm:ss[.fff[fff]]” format (regardless of the precision of the timestamp SQL data type).

SQL to C Data Conversion Examples

Table 63. 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_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0 [a]	00000
SQL_DATE	1992-12-31	SQL_C_CHAR	10	---	22003
SQL_DATE	1992-12-31	SQL_C_TIMESTAMP	ignored	1992,12,31, 0,0,0,0 [b]	00000
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 [a]	00000
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 [a]	01004
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	---	22003

Note:

[a] "\0" represents a null termination character.

[b] The numbers in this list are the numbers stored in the fields of the TIMESTAMP_STRUCT structure.

SQLSTATE **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

Converting Data from C to SQL Data Types

The tables in the following sections describe how the driver or data source converts data sent to the data source. For a given C data type:

- The first column of the table lists the legal input values of the *fSqlType* argument in **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 **SQLSetParam**, which the driver performs to determine if it can convert the data.
- The third column lists the SQLSTATE returned for each outcome by **SQLExecDirect** or **SQLExecute**.

  Note: Data is sent to the data source only if the SQLSTATE is 00000 (Success).

C to SQL Data Types

The tables list the conversions that are defined by ODBC to be valid for a given SQL data type.

If the *fSqlType* argument in **SQLSetParam** contains a value not shown in the table for a given C data type, **SQLSetParam** returns SQLSTATE 07006 (Restricted data type attribute violation).

If the *fSqlType* argument contains a value that is shown in the table but that specifies a conversion not supported by the driver, **SQLSetParam** returns SQLSTATE S1C00 (Driver not capable).

If the *rgbValue* and *pcbValue* arguments specified in **SQLSetParam** are both null pointers, that function returns SQLSTATE S1009 (Invalid argument value).

Length of data
The total length of the data after it has been converted to the specified SQL data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is sent to the data source.

Column length
The maximum number of bytes returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte.

Display size
The maximum number of bytes needed to display data in character form.

Significant digits
The minus sign (if needed) and the digits to the left of the decimal point.

Converting Character C Data to SQL Data

The character C data type is:

SQL_C_CHAR

Table 64. Converting Character C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Length of data <= Column length	00000
	Length of data > Column length	22001
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_REAL SQL_FLOAT SQL_DOUBLE	Data converted without truncation	00000
	Data converted with truncation, but without loss of significant digits	22001
	Conversion of data would result in loss of significant digits	22003
	Data value is not a numeric value	22005
SQL_DATE	Data value is a valid date	00000
	Data value is not a valid date	22008
SQL_TIME	Data value is a valid time	00000
	Data value is not a valid time	22008
SQL_TIMESTAMP	Data value is a valid timestamp	00000
	Data value is not a valid timestamp	22008
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC	Length of data / 2 <= Column length	00000
	Length of data / 2 < Column length	22001

Note:

SQLSTATE **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

Converting Numeric C Data to SQL Data

The numeric C data types are:

SQL_C_SHORT
SQL_C_LONG
SQL_C_FLOAT
SQL_C_DOUBLE

C to SQL Data Types

Table 65. Converting Numeric C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_REAL SQL_FLOAT SQL_DOUBLE	Data converted without truncation	00000
	Data converted with truncation, but without loss of significant digits	22001
	Conversion of data would result in loss of significant digits	22003
SQL_CHAR SQL_VARCHAR	Data converted without truncation.	00000
	Conversion of data would result in loss of significant digits.	22003

Note:

SQLSTATE **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

Converting Date C Data to SQL Data

The date C data type is:

SQL_C_DATE

Table 66. Converting Date C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 10	00000
	Column length < 10	22003
SQL_DATE	Data value is a valid date	00000
	Data value is not a valid date	22007

Note:

SQLSTATE **00**000 is not returned by `SQLError()`; 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_TIME

Table 67. Converting Time C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 8	00000
	Column length < 8	22003
SQL_TIME	Data value is a valid time	00000
	Data value is not a valid time	22008

Note:

SQLSTATE **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

Converting Timestamp C Data to SQL Data

The timestamp C data type is:

SQL_C_TIMESTAMP

Table 68. Converting Timestamp C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= Display size	00000
	19 <= Column length < Display size[a]	22001
	Column length < 19	22003
SQL_DATE	Data value is a valid date[b]	22001
	Data value is not a valid date	22007
SQL_TIME	Data value is a valid time[c]	22001
	Data value is not a valid time	22008
SQL_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 **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

C to SQL Data Conversion Examples

C to SQL Data Types

Table 69. C to SQL Data Conversion Examples

C Data Type	C Data Value	SQL Data Type	Column length	SQL Data Value	SQL STATE
SQL_C_CHAR	abcdef\0	SQL_CHAR	6	abcdef	00000
SQL_C_CHAR	abcdef\0	SQL_CHAR	5	abcde	22001
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6	1234.56	00000
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	5	1234.5	22001
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	3	---	22003
SQL_C_FLOAT	1234.56	SQL_FLOAT	not applicable	1234.56	00000
SQL_C_FLOAT	1234.56	SQL_INTEGER	not applicable	1234	22001

Note:

SQLSTATE **00**000 is not returned by `SQLError()`; rather, it is indicated when the function returns SQL_SUCCESS.

Appendix C. Command Line Interface Include File

This is a common file used by the API to describe function calls and structures and constants for the API. It is called SQLCLI.H and is found in <your instance path>/sqllib/include.

```
/******************************************************************************
 *
 * Source File Name = SQLCLI.H  1.19
 * engn/cli/sqlcli.h, engn, engn_drv_aix
 *
 * (C) COPYRIGHT International Business Machines Corp. 1993
 * All Rights Reserved
 * Licensed Materials - Property of IBM
 *
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
 *
 * Function = Include File defining:
 *              CLI Interface - Constants
 *              CLI Interface - Data Structures
 *              CLI Interface - Function Prototypes
 *
 * Operating System = Common C Include File
 *
 ******************************************************************************/

#ifndef SQL_H_SQLCLI
   #define SQL_H_SQLCLI            /* Permit duplicate Includes */

#include "sqlsystm.h"              /* System dependent defines  */

/* generally useful constants */
#define  SQL_NTS                  -3  /* NTS = Null Terminated String    */
#define  SQL_SQLSTATE_SIZE         5  /* size of SQLSTATE, not including
                                         null terminating byte           */
#define  SQL_MAX_MESSAGE_LENGTH  1024 /* message buffer size             */
#define  SQL_MAX_DSN_LENGTH       32  /* maximum data source name size   */
#define  SQL_MAX_ID_LENGTH        18  /* maximum identifier name size,
                                         e.g. cursor names               */

/* RETCODE values             */
#define  SQL_SUCCESS             0
#define  SQL_SUCCESS_WITH_INFO   1
#define  SQL_NO_DATA_FOUND     100
#define  SQL_NO_DATA           SQL_NO_DATA_FOUND
#define  SQL_ERROR              -1
#define  SQL_INVALID_HANDLE     -2

/* SQLFreeStmt option values  */
#define  SQL_CLOSE               0
```

```
#define  SQL_DROP                1
#define  SQL_UNBIND              2
#define  SQL_RESET_PARAMS        3

/* SQLSetParam defines        */
#define  SQL_C_DEFAULT          99

/* SQLTransact option values  */
#define  SQL_COMMIT              0
#define  SQL_ROLLBACK            1

/* Standard SQL data types */
#define  SQL_CHAR                1
#define  SQL_NUMERIC             2
#define  SQL_DECIMAL             3
#define  SQL_INTEGER             4
#define  SQL_SMALLINT            5
#define  SQL_FLOAT               6
#define  SQL_REAL                7
#define  SQL_DOUBLE              8
#define  SQL_DATE                9
#define  SQL_TIME               10
#define  SQL_TIMESTAMP          11
#define  SQL_VARCHAR            12

/* SQL extended data types */
#define  SQL_LONGVARCHAR        -1
#define  SQL_BINARY             -2  /* Not supported */
#define  SQL_VARBINARY          -3  /* Not supported */
#define  SQL_LONGVARBINARY      -4  /* Not supported */
#define  SQL_BIGINT             -5  /* Not supported */
#define  SQL_TINYINT            -6  /* Not supported */
#define  SQL_BIT                -7  /* Not supported */
#define  SQL_GRAPHIC            -8
#define  SQL_VARGRAPHIC         -9
#define  SQL_LONGVARGRAPHIC    -10

/* C data type to SQL data type mapping */
#define  SQL_C_CHAR       SQL_CHAR      /* CHAR, VARCHAR, DECIMAL, NUMERIC */
#define  SQL_C_LONG       SQL_INTEGER   /* INTEGER                         */
#define  SQL_C_SHORT      SQL_SMALLINT  /* SMALLINT                        */
#define  SQL_C_FLOAT      SQL_REAL      /* REAL                            */
#define  SQL_C_DOUBLE     SQL_DOUBLE    /* FLOAT, DOUBLE                   */
#define  SQL_C_DATE       SQL_DATE      /* DATE                            */
#define  SQL_C_TIME       SQL_TIME      /* TIME                            */
#define  SQL_C_TIMESTAMP  SQL_TIMESTAMP /* TIMESTAMP                       */
#define  SQL_C_BINARY     SQL_BINARY    /* Not supported                   */
#define  SQL_C_BIT        SQL_BIT       /* Not supported                   */
#define  SQL_C_TINYINT    SQL_TINYINT   /* Not supported                   */

/*
 * NULL status defines; these are used in SQLColAttributes, SQLDescribeCol,
 * to describe the nullability of a column in a table.
 */

#define  SQL_NO_NULLS     0
#define  SQL_NULLABLE     1

/* Special length values  */
#define  SQL_NULL_DATA   -1


/* NULL handle defines    */
#define  SQL_NULL_HENV                0L
#define  SQL_NULL_HDBC                0L
#define  SQL_NULL_HSTMT               0L

/* SQL portable types for C  */
#ifdef DB2WIN
typedef           unsigned char         UCHAR;
typedef           signed   char         SCHAR;
typedef           long     int          SDWORD;
typedef           short    int          SWORD;
typedef  unsigned long     int          UDWORD;
typedef  unsigned short    int          UWORD;
typedef                    double       SDOUBLE;
typedef                    float        SFLOAT;


typedef  void  FAR    *    PTR;
typedef  void  FAR    *    HENV;
typedef  void  FAR    *    HDBC;
typedef  void  FAR    *    HSTMT;

typedef  int               RETCODE;


/* The following are provided to enhance portability to/from WINDOWS */
typedef  UCHAR             SQLCHAR;
typedef  SDWORD            SQLINTEGER;
typedef  SWORD             SQLSMALLINT;
typedef  SDOUBLE           SQLDOUBLE;
typedef  SFLOAT            SQLREAL;
typedef  PTR               SQLPOINTER;
typedef  HENV              SQLHENV;
typedef  HDBC              SQLHDBC;
typedef  HSTMT             SQLHSTMT;
typedef  RETCODE           SQLRETURN;



typedef  struct tagDATE_STRUCT
  {
    SWORD   year;
    UWORD   month;
    UWORD   day;
  } DATE_STRUCT;

typedef  struct tagTIME_STRUCT
  {
    UWORD   hour;
    UWORD   minute;
    UWORD   second;
  } TIME_STRUCT;

typedef  struct tagTIMESTAMP_STRUCT
  {
    SWORD   year;
    UWORD   month;
    UWORD   day;
    UWORD   hour;
    UWORD   minute;
    UWORD   second;
    UDWORD  fraction;     /* fraction of a second */
  } TIMESTAMP_STRUCT;


#else

typedef  unsigned char     SQLCHAR;
typedef  long     int      SQLINTEGER;
typedef  short    int      SQLSMALLINT;
typedef  double            SQLDOUBLE;
typedef  float             SQLREAL;

typedef  void *            PTR;
typedef  PTR               SQLPOINTER;
typedef  long              HENV;
typedef  long              HDBC;
typedef  long              HSTMT;
typedef  HENV              SQLHENV;
typedef  HDBC              SQLHDBC;
typedef  HSTMT             SQLHSTMT;

typedef  SQLINTEGER        RETCODE;
typedef  RETCODE           SQLRETURN;



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

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

typedef  struct TIMESTAMP_STRUCT
  {
    SQLSMALLINT   year;
    SQLSMALLINT   month;
    SQLSMALLINT   day;
    SQLSMALLINT   hour;
    SQLSMALLINT   minute;
    SQLSMALLINT   second;
    SQLINTEGER    fraction;     /* fraction of a second */
  } TIMESTAMP_STRUCT;


#endif



/* Core Function Prototypes  */

#ifdef DB2WIN

RETCODE   SQL_API_FN  SQLAllocConnect  (HENV              henv,
                                        HDBC   FAR       *phdbc);

RETCODE   SQL_API_FN  SQLAllocEnv      (HENV   FAR       *phenv);

RETCODE   SQL_API_FN  SQLAllocStmt     (HDBC              hdbc,
                                        HSTMT  FAR       *phstmt);

RETCODE   SQL_API_FN  SQLBindCol       (HSTMT             hstmt,
                                        UWORD             icol,
                                        SWORD             fCType,
                                        PTR               rgbValue,
                                        SDWORD            cbValueMax,
                                        SDWORD FAR       *pcbValue);


RETCODE   SQL_API_FN  SQLConnect       (HDBC              hdbc,
                                        UCHAR  FAR       *szDSN,
                                        SWORD             cbDSN,
                                        UCHAR  FAR       *szUID,
                                        SWORD             cbUID,
                                        UCHAR  FAR       *szAuthStr,
                                       SWORD             cbAuthStr);

RETCODE   SQL_API_FN  SQLDescribeCol   (HSTMT             hstmt,
                                        UWORD             icol,
                                        UCHAR  FAR       *szColName,
                                        SWORD             cbColNameMax,
                                        SWORD  FAR       *pcbColName,
                                        SWORD  FAR       *pfSqlType,
                                        UDWORD FAR       *pcbColDef,
                                        SWORD  FAR       *pibScale,
                                        SWORD  FAR       *pfNullable);

RETCODE   SQL_API_FN  SQLDisconnect    (HDBC              hdbc);

RETCODE   SQL_API_FN  SQLError         (HENV              henv,
                                        HDBC              hdbc,
                                        HSTMT             hstmt,
                                        UCHAR  FAR       *szSqlState,
                                        SDWORD FAR       *pfNativeError,
                                        UCHAR  FAR       *szErrorMsg,
                                        SWORD             cbErrorMsgMax,
                                        SWORD  FAR       *pcbErrorMsg);

RETCODE   SQL_API_FN  SQLExecDirect    (HSTMT             hstmt,
                                        UCHAR  FAR       *szSqlStr,
                                        SDWORD            cbSqlStr);

RETCODE   SQL_API_FN  SQLExecute       (HSTMT             hstmt);

RETCODE   SQL_API_FN  SQLFetch         (HSTMT             hstmt);

RETCODE   SQL_API_FN  SQLFreeConnect   (HDBC              hdbc);

RETCODE   SQL_API_FN  SQLFreeEnv       (HENV              henv);

RETCODE   SQL_API_FN  SQLFreeStmt      (HSTMT             hstmt,
                                        UWORD             fOption);

RETCODE   SQL_API_FN  SQLGetCursorName (HSTMT             hstmt,
                                        UCHAR  FAR       *szCursor,
                                        SWORD             cbCursorMax,
                                        SWORD  FAR       *pcbCursor);


RETCODE   SQL_API_FN  SQLNumResultCols (HSTMT             hstmt,
                                        SWORD  FAR       *pccol);

RETCODE   SQL_API_FN  SQLPrepare       (HSTMT             hstmt,
                                        UCHAR  FAR       *szSqlStr,
                                        SDWORD            cbSqlStr);

RETCODE   SQL_API_FN  SQLRowCount      (HSTMT             hstmt,
                                        SDWORD FAR       *pcrow);
```

```
RETCODE   SQL_API_FN  SQLSetParam      (HSTMT             hstmt,
                                        UWORD             ipar,
                                        SWORD             fCType,
                                        SWORD             fSqlType,
                                        UDWORD            cbParamDef,
                                        SWORD             ibScale,
                                        PTR               rgbValue,
                                        SDWORD FAR       *pcbValue);

RETCODE   SQL_API_FN  SQLTransact      (HENV              henv,
                                        HDBC              hdbc,
                                        UWORD             fType);



#else



SQLRETURN SQL_API_FN  SQLAllocConnect  (SQLHENV              henv,
                                        SQLHDBC             *phdbc);


SQLRETURN SQL_API_FN  SQLAllocEnv      (SQLHENV             *phenv);

SQLRETURN SQL_API_FN  SQLAllocStmt     (HDBC              hdbc,
                                        HSTMT            *phstmt);

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


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

SQLRETURN SQL_API_FN  SQLDescribeCol   (SQLHSTMT             hstmt,
                                        SQLSMALLINT       icol,
                                        SQLCHAR          *szColName,
                                        SQLSMALLINT       cbColNameMax,
                                        SQLSMALLINT      *pcbColName,
                                        SQLSMALLINT      *pfSqlType,
                                        SQLINTEGER       *pcbColDef,
                                        SQLSMALLINT      *pibScale,
                                        SQLSMALLINT      *pfNullable);

SQLRETURN SQL_API_FN  SQLDisconnect    (SQLHDBC              hdbc);

SQLRETURN SQL_API_FN  SQLError         (SQLHENV              henv,
                                        SQLHDBC              hdbc,
                                        SQLHSTMT             hstmt,
                                        SQLCHAR          *szSqlState,
                                        SQLINTEGER       *pfNativeError,
                                        SQLCHAR          *szErrorMsg,
                                        SQLSMALLINT       cbErrorMsgMax,
                                        SQLSMALLINT      *pcbErrorMsg);

SQLRETURN SQL_API_FN  SQLExecDirect    (SQLHSTMT             hstmt,
                                        SQLCHAR          *szSqlStr,
                                        SQLINTEGER        cbSqlStr);

SQLRETURN SQL_API_FN  SQLExecute       (SQLHSTMT             hstmt);

SQLRETURN SQL_API_FN  SQLFetch         (SQLHSTMT             hstmt);

SQLRETURN SQL_API_FN  SQLFreeConnect   (SQLHDBC              hdbc);

SQLRETURN SQL_API_FN  SQLFreeEnv       (SQLHENV              henv);

SQLRETURN SQL_API_FN  SQLFreeStmt      (SQLHSTMT             hstmt,
                                        SQLSMALLINT       fOption);

SQLRETURN SQL_API_FN  SQLGetCursorName (SQLHSTMT             hstmt,
                                        SQLCHAR          *szCursor,
                                        SQLSMALLINT       cbCursorMax,
                                        SQLSMALLINT      *pcbCursor);


SQLRETURN SQL_API_FN  SQLNumResultCols (SQLHSTMT             hstmt,
                                        SQLSMALLINT      *pccol);

SQLRETURN SQL_API_FN  SQLPrepare       (HSTMT             hstmt,
                                        SQLCHAR          *szSqlStr,
                                        SQLINTEGER        cbSqlStr);

SQLRETURN SQL_API_FN  SQLRowCount      (SQLHSTMT             hstmt,
                                        SQLINTEGER       *pcrow);

SQLRETURN SQL_API_FN  SQLSetParam      (HSTMT             hstmt,
                                        SQLSMALLINT       ipar,
                                        SQLSMALLINT       fCType,
                                        SQLSMALLINT       fSqlType,
                                        SQLINTEGER        cbParamDef,
                                        SQLSMALLINT       ibScale,
                                        SQLPOINTER        rgbValue,
                                        SQLINTEGER       *pcbValue);
```

```
SQLRETURN SQL_API_FN  SQLTransact      (SQLHENV              henv,
                                         SQLHDBC              hdbc,
                                         SQLSMALLINT       fType);


#endif

#endif /* SQL_H_SQLCLI */
```

Appendix D. State Transition Tables

The following tables show the effect of each generic access API function on the states of the environment (*henv*), connection (*hdbc*) and statement (*hstmt*) handles.

Each entry in the tables is the result state, or set of result states, of the handle after execution of the function. Unless noted, an error from a function causes no state transition.

The environment can be in one of the following states:

- S0 unallocated environment
- S1 allocated environment
- S2 allocated *hdbc*
- S3 connected *hdbc*

Table 70 lists the next valid state for each function when called from the given state. “IH” indicates an INVALID_HANDLE return code.

Table 70. Environment and Connection State Transitions

Function	S0	S1	S2	S3
SQLAllocEnv	S1	S1	S1	S1
SQLAllocConnect	IH	S2	S2	S2
SQLConnect	IH	IH	S3	*08002*
SQLDisconnect	IH	IH	*08003*	S2 *25000*
SQLFreeConnect	IH	IH	S1	*S1010*
SQLFreeEnv	IH	S0	*S1010*	*S1010*
SQLTransact	IH	IH	*08003*	S3

A statement handle (*hstmt*) can be in one of the following states:

- S0 Not allocated
- S1 Allocated
- S2 Prepared
- S3 Executed, or cursor open but not positioned on a row
- S4 Cursor positioned on a row

DataJoiner must establish a successful connection before allocating a statement. Table 70 lists the next valid state for each function when called from a given state. “IH” indicate and INVALID_HANDLE return code.

Table 71 (Page 1 of 2). Statement Transitions

Function	S0	S1	S2	S3	S4
SQLAllocStmt	S1	S1 (1)	S1 (1)	S1 (1)	S1 (1)

Table 71 (Page 2 of 2). Statement Transitions

Function	S0	S1	S2	S3	S4
SQLBindCol	IH	S1	S2	S3	S4
SQLDescribeCol	IH	*S1010*	S2 (2) *24000*(3)	S3 (2) *24000*(3)	S4 (2) *24000*(3)
SQLDisconnect (4)	S0	S0	S0	S0 (5)	S0 (5)
SQLExecDirect (6)	IH	S3 (7)	S3 (7)	S3 (8)	*24000*
SQLExecute (6)	IH	*S1010*	S3 (7)	S3 (8)	*24000*
SQLFetch	IH	*S1010*	*S1010*	S3 (9,10), S4, *24000* (3)	S3 (9), S4
SQLFreeStmt (11)	IH	S1	S2	S1 (12), S2 (13)	S1 (12), S2 (13)
SQLFreeStmt (14)	IH	S0	S0	S0	S0
SQLFreeStmt (15)	IH	S1	S2	S3	S4
SQLGetCursorName	IH	*S1015*	*S1015*	S3 *S1015*	S4
SQLNumResultCols	IH	*S1010*	S2	S3	S4
SQLPrepare (16)	IH	S2	S1 (17), S2 (8)	S2	*24000*
SQLRowCount	IH	*S1010*	*S1010*	S3 (18)	S4 (18)
SQLSetParam	IH	S1	S2	S3 (19)	S4 (19)
SQLTransact (4)	S0	S1	S1, S2 (20)	S1, S3 (20)	S1, S3, S4 (20)

Notes

1. Allocation functions should never be called for allocated handles, as the data access module will loss any information associated with the handle and the handle will return to the allocated state, S1.
2. Occurs when the executed statement created a result set.
3. Occurs when the executed statement did not create a result set.
4. Transition for all *hstmt*s that are allocated with SQLAllocStmt for the same *hdbc*.
5. SQLDisconnect returns a SQLSTATE of 25000 (Invalid transaction state) if there is an incomplete transaction on an *hstmt* associated with the *hdbc*.
6. The *hstmt* indicated by the cursor in UPDATE WHERE CURRENT OF cursor or DELETE WHERE CURRENT OF cursor must be in state S4, or the SQLExecute or SQLExecDirect function returns SQL_ERROR with an SQLSTATE of 24000 (Invalid cursor state). Following the positioned UPDATE or DELETE, the positioned *hstmt* is left in state S4.

7. If the SQL statement associated with the *hstmt* contains parameters, and one or more of the parameters have not been set, the data access module should return SQL_ERROR with SQLSTATE 07001 (Wrong number of parameters).
8. This transition is legal only when there are no open cursors on the *hstmt*.
9. Transition for SQL_NO_DATA_FOUND.
10. Transition for an executed statement that was not a SELECT. SQLFetch returns SQL_ERROR with an SQLSTATE 24000.
11. Transition for *fOption* equal to SQL_CLOSE.
12. Perform the transition if the query was not executed with SQLExecute. (The execution was done by SQLExecDirect or a catalog function.)
13. Perform the transition if the query was executed with SQLExecute.
14. Transition for *fOption* equal to SQL_DROP.
15. Transition for *fOption* equal to SQL_UNBIND or SQL_RESET_PARAMS.
16. The *hstmt* indicated by cursor in WHERE CURRENT OF cursor must be in state S3 or S4 for SQLPrepare. After SQLPrepare, the *hstmt* remains in the same state.
17. Transition where SQLPrepare is called for an *hstmt* that is already in the S2 state and SQLPrepare fails for a reason other than validation.
18. Number of rows returned for INSERT, UPDATE, and DELETE statements if available; otherwise, the data access module should return a -1 for row count indicating number of rows not available or SQLRowCount is undefined for the SQL statement.
19. Resetting parameters has no effect on the executed statement.
20. Transitions for data sources that can retain cursor state across transaction boundaries.

Appendix E. Incompatibilities between Versions of DataJoiner

This appendix identifies the incompatibilities between DataJoiner version 1 and DataJoiner version 2. We will discuss changes in system catalog tables and in functionality, the symptoms that result when you try to use tables and functionality that have changed, and ways to avoid or counteract these symptoms.

Definition of Incompatibility

Some tables and processes in DataJoiner version 2 are incompatible with their version 1 counterparts. Thus, if an existing application uses any of them, results might be different than expected, an error might occur, or performance might be reduced. In this context, the term *application* applies to a broad range of things; for example:

- Application program code
- Third-party utilities
- Interactive SQL queries
- Command and/or API invocation

This appendix does not describe:

- Operations that in version 2 are less likely to generate an error than they did in version 1. Such operations will have only a positive impact on existing applications.
- Incompatibilities common to DataJoiner and DB2. For these incompatibilities, see the *DB2 SQL Reference.*

System Catalog Tables

This section discusses problems that you would encounter if you use applications based on version 1 to query and modify version 2 system catalog tables.

Columns and Values in System Catalog Tables

Change

Changes have been made to columns, column attributes, and the acceptable values in several DataJoiner system catalog tables. This section discusses changes that could cause problems for applications designed to use the version 1 form of the tables. To ascertain other changes, you might query the version 1 tables and the version 2 system catalog views that correspond to them, or you could read descriptions of these tables and views. The tables are described in the version 1.2 edition of the *Application Programming and SQL Reference Supplement.* The views are described in the version 2 edition of this book.

SYSCOLUMNS

HIGH2KEY: Non-character values are now in printable format rather than binary format.

LOW2KEY: Non-character values are now in printable format rather than binary format.

NULLS: The value D (not null with default) has been changed to N (not nullable).

REMOTE_TYPE: In version 1, values denoted data types of columns of data source tables that DataJoiner referenced by nickname. In version 2, these values are stored in REMOTE_TYPENAME.

SYSINDEXES

CLUSTERRATIO: In version 1, the value in this column was -1 if statistics were not gathered. In version 2, the value is -1 either if statistics are not gathered or if detailed index statistics are gathered. In the latter case, an appropriate value is added to the CLUSTERFACTOR column.

SYSSERVERS

COLSEQ: Deleted from SYSSERVERS. In version 2, this server option is denoted by a value (colseq) in the OPTION column of the SYSCAT.SERVER_OPTIONS catalog view.

CONNECTSTRING: Deleted from SYSSERVERS. In version 2, this server option is denoted by a value (connectstring) in the OPTION column of the SYSCAT.SERVER_OPTIONS catalog view.

CPURATIO: Data type changed from DOUBLE to FLOAT.

DATEFORMAT: Deleted from SYSSERVERS. In version 2, this server option is denoted by a value (DATEFORMAT) in the OPTION column of the SYSCAT.SERVER_OPTIONS catalog view.

FOLDID: Deleted from SYSSERVERS. In version 2, this server option is denoted by a value (fold_id) in the OPTION column of the SYSCAT.SERVER_OPTIONS catalog view.

IORATIO: Data type changed from DOUBLE to FLOAT.

PASSWORD: Deleted from SYSSERVERS. In version 2, this server option is denoted by a value (password) in the OPTION column of the SYSCAT.SERVER_OPTIONS catalog view.

TIMEFORMAT: Deleted from SYSSERVERS. In version 2, this server option is denoted by a value (TIMEFORMAT) in the OPTION column of the SYSCAT.SERVER_OPTIONS catalog view.

TIMESTAMPFORMAT: Deleted from SYSSERVERS. In version 2, this server option is denoted by a value (TIMESTAMPFORMAT) in the OPTION column of the SYSCAT.SERVER_OPTIONS catalog view.

SYSREMOTEUSERS

AUTHID Data type changed from CHAR to VARCHAR.

SYSTABLES

PACKED_DESC: Data type changed from LONGVARCHAR to BLOB.

REL_DESC: Data type changed from LONGVARCHAR to BLOB.

VIEW_DESC: Data type changed from LONGVARCHAR to BLOB.

Symptom

Obviously, a variety of symptoms could occur.

If an existing application does a qualified search on a column that takes a different value than it did before (for example, a search on NULLS in SYSIBM.SYSCOLUMNS for a value of D), the application might react differently than expected.

If an existing application accesses a column whose data type has changed (for example, CPURATIO in SYSIBM.SYSSERVERS), you might retrieve too little data or too much.

Resolution

Review the changes listed above to decide whether they affect your applications and, if so, what corrective action to take (for example, updating the application). So that any problems in accessing or maintaining catalog table data can be avoided, we strongly recommend that you query the version 2 catalog views instead of the tables.

If you need a rough approximation of the degree of clustering, select both CLUSTERRATIO and CLUSTERFACTOR in the SYSCAT.INDEXES catalog view and choose the greater of the two values that you retrieve.

How Users Modify System Catalog Tables

Change

For DataJoiner to perform operations on a specific data source, DataJoiner must associate an identifier (specifically, a server name) with that data source. In version 1, you could create such an association by INSERTing appropriate values into the table SYSIBM.SYSSERVERS. You could also modify an association by UPDATing SYSIBM.SYSSERVERS, and terminate an association by deleting a server name from SYSIBM.SYSSERVERS. In version 2, you use DDL to perform these same operations indirectly. Specifically, you create DataJoiner-to-data source associations with the CREATE SERVER MAPPING statement, modify them with the ALTER SERVER MAPPING statement, and terminate them with the DROP statement. These statements operate on SYSCAT.SYSSERVERS, a catalog view derived from SYSIBM.SYSSERVERS. The changes that you make to the view are propagated to SYSIBM.SYSSERVERS.

For a user to access data sources from DataJoiner, DataJoiner must associate the ID under which the user connects to DataJoiner with the IDs under which the user connects to these data sources. In version 1, you could create such an association by INSERTing appropriate values into the table SYSIBM.SYSREMOTEUSERS. You could also modify an association by UPDATing SYSIBM.SYSREMOTEUSERS, and terminate an association by deleting an ID from SYSIBM.REMOTEUSERS. In version 2, you use DDL to perform these same operations indirectly. Specifically, you create associations between IDs with the CREATE USER MAPPING statement, modify them with the ALTER USER MAPPING statement, and terminate them with the DROP statement. These statements operate on SYSCAT.REMOTEUSERS, a catalog view derived from SYSIBM.REMOTEUSERS. The changes that you make to the view are propagated to SYSIBM.REMOTEUSERS.

Symptom

If you issue an INSERT, UPDATE, or DELETE statement against SYSIBM.SYSSERVERS, SYSIBM.REMOTEUSERS, or any of DataJoiner's other system catalog tables, the statement will fail.

Resolution

To modify SYSIBM.SYSSERVERS or SYSIBM.REMOTEUSERS, use the SERVER MAPPING or USER MAPPING DDLs, as described in “Change” on page 129.

Appendix F. DataJoiner Education and Service

This section describes the customer education courses and the types of assistance available for DataJoiner.

DataJoiner Education

IBM offers customer education courses that teach you how to install, use, and maintain DataJoiner. The courses are described in this section.

For more information, or to enroll in any IBM class, call 1-800-IBM-TEACH (1-800-426-8322) and refer to the IBM US Course Code. For locations outside the United States, contact your IBM representative.

Course descriptions will also be maintained at the DataJoiner web site. The DataJoiner URL is:

```
http://www.software.ibm.com/data/datajoiner/
```

Using DataJoiner

IBM US Course Code U4253, World-Wide Code DW20

Duration 2 days

Format Lecture with classroom exercises.

This course introduces the student to DataJoiner and its powerful multidatabase server capabilities. After completing this course, students should be able to effectively use DataJoiner to perform simple and complex distributed requests. They should also be able to monitor and tune SQL queries, accounting for the capabilities and characteristics of diverse DataJoiner data sources. Areas covered include:

- Global optimization
- Multi-vendor query considerations
- Nicknames
- Basic security
- An introduction to the DataJoiner catalog
- DataJoiner query performance
- The DataJoiner Explain tool
- The DataJoiner Database System Monitor

Who Should Take This Course

This course is appropriate for anyone who will be using, managing, installing, or maintaining a DataJoiner multiple database environment.

Prerequisite

SQL experience. You can obtain this experience by attending the “SQL Workshop,” IBM US Course Code U4045.

DataJoiner Administration

IBM US Course Code U4254, World-Wide Code DW21

Duration 3 days

Format Lecture with classroom exercises.

This course trains the student to install, configure, and manage a secure DataJoiner multidatabase server environment. Areas covered include:

- Installing DataJoiner
- Generating and managing the DataJoiner database
- Configuring DataJoiner
- Enabling DataJoiner client access to remote data sources
- DataJoiner security
- DataJoiner server performance

Who Should Take This Course

This course is appropriate for anyone who will be managing, installing, or maintaining a DataJoiner multiple database environment.

Prerequisite

DataJoiner knowledge or experience. You can obtain this experience by attending "Using DataJoiner," IBM US Course Code U4253.

DataJoiner Service Providers

IBM provides services for DataJoiner that include assistance with planning, installing, and configuring the product. The assistance is customized to your individual environment and takes place in two phases.

First Phase: Planning

The first phase helps you plan for the installation and configuration of DataJoiner, and configure network systems so DataJoiner can communicate optimally with all data sources and clients. It includes:

- Assessing general readiness
- Defining clients
- Defining data sources
- Assessing applications
- Defining backup and recovery strategies for DataJoiner
- Configuring DataJoiner database parameters
- Identifying test queries for system validation
- Defining security requirements

Second Phase: Implementation

The second phase focuses on implementation of the plan developed in the planning phase described above. It includes:

- Installing DataJoiner
- Configuring data sources
- Providing access to data source tables and views

- Installing and configuring remote clients
- Validating and documenting the environment
- Providing final turnover to the customer

At the end of this phase, active remote and local clients can access multiple data sources through DataJoiner.

DataJoiner services can be combined with replication services if you are interested in replicating data across a heterogeneous database environment. For more information about DataJoiner and replication services, contact your IBM representative or see the DataJoiner web page. The DataJoiner URL is:

```
http://www.software.ibm.com/data/datajoiner/
```

Index

We'd Like to Hear from You

DB2 DataJoiner
Generic Access API Reference
Version 2 Release 1

Publication No. SC26-9147-00

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773 or (408) 463-4393.
- Electronic mail—Use the following network ID:
  - Internet: COMMENTS@VNET.IBM.COM

  Be sure to include the following with your comments:

  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

DB2 DataJoiner
Generic Access API Reference
Version 2 Release 1

Publication No. SC26-9147-00

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□
Grammatically correct and consistent	□	□	□	□	□
Graphically well designed	□	□	□	□	□
Overall satisfaction	□	□	□	□	□

Please tell us how we can improve this book:

May we contact you to discuss your comments? □ Yes □ No

Name

Address

Company or Organization

Phone No.

Readers' Comments
SC26-9147-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department W92/H3
PO Box 49023
San Jose, CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC26-9147-00

Spine information:

IBM DB2 DataJoiner Generic Access API Reference *Version 2 Release 1*