IBM solidDB
IBM solidDB Universal Cache
Version 6.3

Programmer Guide

SC23-9825-03

Note

Before using this information and the product it supports, read the information in "Notices" on page 287.

First edition, third revision

This edition applies to version 6, release 3 of IBM solidDB (product number 5724-V17) and IBM solidDB Universal Cache (product number 5724-W91) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

Summary of changes

Changes for revision 03

- Information about the solidDB® Light Client has been removed; solidDB Light Client is deprecated as of 6.3 Fix Pack 8.

Changes for revision 02

- The following Optimizer hints have been added in section “Hints” on page 22:
  - TRIPLE MERGE JOIN
  - UNION FOR OR
  - OR FOR OR
  - LOOP FOR OR

Changes for revision 01

- Editorial corrections.

About this manual

This guide contains information about using IBM® solidDB® through the different Application Programming Interfaces, with or without the solidDB linked library access and HotStandby.

solidDB ODBC Driver, solidDB Light Client, and solidDB JDBC Driver help your client application access solidDB. solidDB's 32-bit native ODBC Driver conforms to the Microsoft® ODBC 3.51 API standard. solidDB Light Client is a lightweight version of the solidDB ODBC API and is intended for environments where the footprint of the client application must be very small. The solidDB JDBC Driver is a solidDB implementation of the JDBC 2.0 standard.

This guide assumes general knowledge of relational databases and SQL. It also assumes familiarity with solidDB. If you will use the ODBC driver, this manual assumes a working knowledge of the C programming language. If you will use the JDBC driver, this manual assumes a working knowledge of the Java™ programming language.

Typographic conventions

solidDB documentation uses the following typographic conventions:

Table 1. Typographic conventions

Format	Used for
Database table	This font is used for all ordinary text.
NOT NULL	Uppercase letters on this font indicate SQL keywords and macro names.
`solid.ini`	These fonts indicate file names and path expressions.
`SET SYNC MASTER YES;` `COMMIT WORK;`	This font is used for program code and program output. Example SQL statements also use this font.
run.sh	This font is used for sample command lines.
TRIG_COUNT()	This font is used for function names.
java.sql.Connection	This font is used for interface names.
LockHashSize	This font is used for parameter names, function arguments, and Windows® registry entries.
argument	Words emphasized like this indicate information that the user or the application must provide.
Administrator Guide	This style is used for references to other documents, or chapters in the same document. New terms and emphasized issues are also written like this.

Table 1. Typographic conventions (continued)

Format	Used for
File path presentation	Unless otherwise indicated, file paths are presented in the UNIX® format. The slash (/) character represents the installation root directory.
Operating systems	If documentation contains differences between operating systems, the UNIX format is mentioned first. The Microsoft Windows format is mentioned in parentheses after the UNIX format. Other operating systems are separately mentioned. There may also be different chapters for different operating systems.

Syntax notation conventions

solidDB documentation uses the following syntax notation conventions:

Table 2. Syntax notation conventions

Format	Used for
`INSERT INTO` *`table_name`*	Syntax descriptions are on this font. Replaceable sections are on *this* font.
`solid.ini`	This font indicates file names and path expressions.
[ ]	Square brackets indicate optional items; if in bold text, brackets must be included in the syntax.
|	A vertical bar separates two mutually exclusive choices in a syntax line.
{ }	Curly brackets delimit a set of mutually exclusive choices in a syntax line; if in bold text, braces must be included in the syntax.
...	An ellipsis indicates that arguments can be repeated several times.
. . .	A column of three dots indicates continuation of previous lines of code.

1 Introduction to IBM solidDB APIs

This section provides an overview of the application programming interfaces available to you for accessing IBM® solidDB databases.

These interfaces enable applications to establish multiple database connections simultaneously and to process multiple SQL statements simultaneously.

solidDB ODBC Driver

The 32-bit native ODBC Driver included in solidDB conforms to the Microsoft ODBC 3.51 API standard.

For differences between the ODBC standard and the solidDB implementation, refer to the appropriate topic in this manual.

On most platforms, the solidDB ODBC Driver is included in thesolidDB Development Kit (SDK).

solidDB provides both a Unicode and an ASCII version of the ODBC driver. For details about the Unicode version, see 3, "Using Unicode," on page 49.

Using solidDB ODBC Driver Functions

Users on all platforms can also access ODBC Driver supported functions with solidDB ODBC API.

The solidDB ODBC API is the native call level interface (CLI) for solidDB databases. It is a distributed in the form of a library (a Dynamic Link Library (DLL) on Microsoft Windows). The solidDB ODBC API is compliant with ANSI X3H2 SQL CLI standard.

solidDB's implementation of ODBC API supports a rich set of database access operations sufficient for creating robust database applications, including:

- Allocating and deallocating handles
- Getting and setting attributes
- Opening and closing database connections
- Accessing descriptors
- Executing SQL statements
- Accessing schema metadata
- Controlling transactions
- Accessing diagnostic information

Depending on the application's request, the solidDB ODBC Driver can automatically commit each SQL statement or wait for an explicit commit or rollback request. When the driver performs a commit or rollback operation, the driver resets all statement requests associated with the connection.

ODBC API basic application steps

A client database application calls the solidDB ODBC API directly (or through the ODBC Driver Manager) to perform all interactions with a database. For example, to insert, delete, update, or select records, you make a series of calls to functions in the ODBC API.

An application using ODBC API performs the following tasks:.

1. The application allocates memory and creates handles, and establishes a connection to the database.
   a. The application allocates memory for an environment handle (henv) and a connection handle (hdbc); both are required to establish a database connection.

      An application may request multiple connections for one or more data sources. Each connection is considered a separate transaction space. In other words, a COMMIT or ROLLBACK on one connection will not commit or rollback any statements executed through any other connection.
   b. The SQLConnect() call establishes the database connection, specifying the server name (a connect string or a data source name), user id, and password.
   c. The application then allocates memory for a statement handle.
2. The application executes the statement. This requires a series of function calls.
   a. The application calls either SQLExecDirect(), which both prepares and executes an SQL statement, or SQLPrepare() and SQLExecute(), which allows statements to be executed multiple times.
   b. If the statement was a SELECT, the result columns must be bound to variables in the application so that the application can see the returned data. The SQLBindCol() function will bind the application's variables to the columns of the result set. The rows can then be fetched using SQLFetch() repeatedly. SELECT statements must be committed as soon as processing of the resultset is done.

      If the statement was an UPDATE, DELETE, or INSERT, then the application needs to check if the execution succeeded and call SQLEndTran() to commit the transaction.
3. Finally the application closes the connection and frees any handles.
   a. The application frees the statement handle.
   b. The application closes the connection.
   c. The application frees the connection and environment handles (hdbc and henv).

Note that step 2 (executing SQL statements) may be done repeatedly, depending upon how many SQL statements need to be executed.

Read 2, “Using solidDB ODBC API,” on page 9 for more information on using these API calls.

Format of the connect string

Connect strings used in ODBC applications follow a common format shown here.

The same format applies also to listen parameters in the configuration file `solid.ini`.

`protocol_name [`*`options`*`] [`*`server_name`*`] [`*`port_number`*`]` where options may be any number of:

Table 3. Connect String Options

Option	Meaning
-z	Data compression is enabled for this connection
-c *milliseconds*	Login timeout is specified (the default is operating-system-specific). A login request fails after the specified time has elapsed. Note: applies only for the TCP protocol.
-r *milliseconds*	Connection (or read) timeout is specified (the default is 60 s). A network request fails when no response is received during the time specified. The value 0 sets the timeout to infinite. Note: applies only for the TCP protocol.

Examples

```
tcp localhost 1315
tcp 1315
tcp -z -c1000 1315
nmpipe host22 SOLIDDB
```

Client-side configuration file

solidDB gets its client configuration information from the client-side `solid.ini` file. The client-side configuration file is used if the ODBC driver is used and the file must be located in the working directory of the application.

Important: In most cases, only solidDB server-side parameters are used when programming for the solidDB. However, occasionally there is a need to use client-side parameters. For example, you may want to create an application that defines no data source, but takes the data source from the connect string in the client-side configuration file.

Note: In solidDB documentation, references to `solid.ini` file are usually for the server-side `solid.ini` file.

When the solidDB is started, it attempts to open the configuration file `solid.ini`. If the file does not exist, solidDB will use the factory values for the parameters. If the file exists, but a value for a particular parameter is not set in the `solid.ini` file, solidDB will use a factory value for that parameter. The factory values may depend on the operating system you are using.

By default, the client looks for the `solid.ini` file in the current working directory, which is normally the directory from which you started the client. When searching for the file, the solidDB uses the following precedence (from high to low):

- location specified by the SOLIDDIR environment variable (if this environment variable is set)
- current working directory

Client-side parameters

This section describes the most important solidDB client-side parameters.

- Com.Connect

The Connect parameter in the [Com] section defines the default network name (connect string) for a client to connect to when it communicates with a server. Since the client should talk to the same network name as the server is listening to, the value of the Connect parameter on the client should match the value of the Listen parameter on the server.

The same format of the connect string applies to all listen configuration parameters as well as to connect strings used in ODBC applications.

- Com.Trace

  If you change the Trace parameter default setting from No to Yes, solidDB starts logging trace information on network messages for the established network connection to the default trace file or to the file specified in the TraceFile parameter.

- Com.TraceFile

  If the Trace parameter is set to Yes, then trace information on network messages is written to a file specified by the TraceFile parameter. If no file name is specified, the server uses the default value `soltrace.out`, which is written to the current working directory of the server or client, depending on which end the tracing is started at.

ODBC non-standard behavior

This section describes the non-standard behavior and limitations of solidDB ODBC driver.

Error information

Regardless of the version set by the client, the driver returns error information based on the ODBC 3.0 specification.

Error in SQLPutData using SQL_NULL_DATA as parameter length

If you try to insert or update one or more data items where one of the items has SQL_NULL_DATA as the length specifier, no data will be inserted. The column value will become NULL.

SQLAllocHandle can return incomplete error information

If you call SQLAllocHandle with an invalid handle type, for example,

```
SQLAllocHandle(-5, hdbc, &hstmt);
```

the function will return SQL_ERROR but not Error State "HY092" or message "Invalid Attribute/Option Identifier".

MSAccess - linking the table with certain column types

After linking the table with data types WCHAR, WVARCHAR, and LONG WVARCHAR, when a user inserts a particular record and then inserts/updates/deletes another record, the driver shows '#deleted' for the previous newly added/updated record.

ADO - OpenSchema methods

The following OpenSchema methods are not supported through ADO:

- adSchemaCatalogs

- adSchemaColumnPrivileges
- adSchemaConstraintColumnUsage
- adSchemaConstraintTableUsage
- adSchemaTableConstraint
- adSchemaForeignKeys
- adSchemaTablePrivileges
- adSchemaViews
- adSchemaViewTableUsage

The above mentioned OpenSchema methods are not supported by ADO with any ODBC Driver. This is a limitation of the Microsoft OLE DB Provider for ODBC. This is NOT specific to the solidDB ODBC Driver.

solidDB JDBC Driver

The JDBC 2.0 Driver provides support for JDBC 2.0.

For solidDB JDBC Driver, Java Development Kit (JDK) 1.4.2 or newer is supported.

solidDB JDBC Driver allows you to develop your application with a Java tool that accesses the database using JDBC. The JDBC API, the core API for JDK 1.2, defines Java classes to represent database connections, SQL statements, result sets, database metadata, and so on. It allows you to issue SQL statements and process the results. JDBC is the primary API for database access in Java.

In order to use JDBC, you have to install the solidDB JDBC Driver. Usage of JDBC drivers varies depending on your Java development environment. Instructions and samples for using the solidDB JDBC Driver are located in the `/jdbc` subdirectory and in 4, “Using the solidDB JDBC Driver,” on page 55.

Building client applications

This section provides an overview of how to create a client application that will work with solidDB. The information in this section applies primarily to C-language programs that use the ODBC driver.

What is a client?

A client application, or "client" for short, is a program that submits requests (SQL queries) to the server and gets results back from the server.

A client program is separate from the server program. In many cases, the client is also running on a separate computer. Using linked library access, you can actually link the client's code directly to the server's code so that both run as a single process. For more information, see *IBM solidDB Linked Library Access User Guide*.

Since the client is a separate program, it cannot directly call functions in the server. Instead, it must use a communications protocol (such as TCP/IP, named pipes, shared memory, etc.) to communicate with the server. Different platforms support different protocols. On some platforms, you may need to link a specific library file (which supports a specific protocol) to your application so that your application can communicate with the server.

How is the query passed to the server?

Queries are written using the SQL programming language.

One way that the server and client can exchange data is simply to pass literal strings back and forth. The client could send the server the string:

```
SELECT name FROM employees WHERE id = 12;
```

and the server could send back the string:

```
"Smith, Jane".
```

In practice, however, communication is usually done via a "driver", such as an ODBC driver or a JDBC driver. "ODBC" stands for "Open DataBase Connectivity" and is an API (Application Programming Interface) designed by Microsoft to make database access more consistent across vendors. If your client program follows the ODBC conventions, then your client program will be able to talk with any database server that follows those same conventions. Most major database vendors support ODBC to at least some extent. The ODBC standard is generally used by programs written in the C programming language.

"JDBC" stands for "Java DataBase Connectivity". It is based heavily on the ODBC standard, and not surprisingly it is essentially "ODBC for Java programs". Information about JDBC is available from the main Java website:

http://java.sun.com/

There are two major ways to pass specific data values (e.g. "Smith, Jane" to the server. The first way is to simply embed the values as literals in the query. You've already seen this in SQL statements like:

```
INSERT INTO employees (id, name) VALUES (12, 'Smith, Jane');
```

This works well if you have a single statement that you want to execute. There are times, however, that you may want to execute the same basic statement with different values. For example, if you want to insert data for 500 employees, you may not want to compose 500 separate statements such as

```
INSERT INTO employees (id, name) VALUES (12, 'Smith, Jane');
INSERT INTO employees (id, name) VALUES (13, 'Jones, Sally');
...
```

Instead, you might prefer to compose a single "generic" statement and then pass specific values for that statement. For example, you might want to compose the following statement:

```
INSERT INTO employees (id, name) VALUES (?, ?);
```

and have the question marks replaced with specific data values. This way you can easily execute all 500 INSERT statements inside a loop without composing a unique INSERT statement for each employee. By using parameters, you can specify different values each time a statement executes. A parameter allows you to specify a variable that will be used by the client program and the ODBC driver to store values that the client and server exchange. In essence, you pass a parameter for each place in the statement where you have a question mark.

Another situation where you might want to use parameters to exchange data values is when working with data that is difficult to represent as string literals. For example, if I want to insert a digitized copy of the song "American Pie" into my database, and I don't want to compose an SQL statement with a literal that contains a series of hexadecimal numbers to represent that digitized data, then I can store the digitized data in an array and notify the ODBC driver of the location of that array.

To use parameters with SQL statements, you go through a multi-step process. Below we describe the process when you are inserting data. The process is somewhat similar when you want to retrieve data.

1. "Prepare" the SQL statement. During the "prepare" phase, the server analyzes the statement and (among other things) looks to see how many parameters there will be. The number and meaning of the parameters is shown by the question marks that are included in the SQL statement.
2. Tell the ODBC driver which variables will be used as parameters. (Telling the ODBC driver which variable is associated with which column or value is called "binding" the parameters)
3. Put values into the parameters (i.e. set the values of the variables).
4. "Execute" the prepared statement.

During the execution phase, the ODBC driver will read the values you have stored in the parameters and will pass those values to the server to use with the statement that it has already prepared.

The process for getting results back is similar, and is described in the next section.

How are the results passed back to the client?

The result of a query is a set of 0 or more rows. If you are using the ODBC driver (or JDBC driver) then you retrieve each row by using the appropriate ODBC (or JDBC) functions.

As a general rule, you go through the following steps

1. "Prepare" the SQL statement. During the "prepare" phase, the server analyzes the statement and (among other things) looks to see how many parameters there will be. The number and meaning of the parameters is shown by the question marks that are included in the SQL statement.
2. Tell the ODBC driver which variables will be used as parameters. (Telling the ODBC driver which variable is associated with which column or value is called "binding" the parameters.)
3. "Execute" the prepared statement. This tells the server to execute the query and collect the result set. Note that the result set is NOT immediately passed to the client, however.
4. "Fetch" the next row of the result set. When you do a "fetch", you tell the server and the ODBC driver to retrieve one row of results from the result set, and then store the values of that row into the parameters that you previously defined for the ODBC driver to share with your application.

Not surprisingly, you will normally perform a loop, fetching one row at a time and reading the data from the parameters after each fetch.

Using the ODBC driver library

The ODBC driver library must be linked with your client application program.

Static vs. dynamic libraries

You will then be able to call the functions that are defined in these libraries. For details about library names, see the Release Notes in the solidDB installation package.

Some library files are static; they are linked to your client application's executable program at the time that you do a compile-and-link operation. Other library files are dynamic; dynamic library files are stored separately from your executable and are loaded into memory at the time your program executes.

The advantage of a static library is that your application is largely self-contained; if you distribute the application to your customers, those customers do not have to install a separate shared library in addition to installing your application.

The advantage of a dynamic library is that on many systems it requires less disk space (and, on some platforms, less memory space) if more than one client uses that library. For example, if you have two client applications that each link to a 5 MB static library, you will need not only 5 MB of disk space to store the static library, but also 10 MB of additional disk space to store both copies of the library that are linked into the application. However, if you link two client applications to a dynamic library, no additional copies of that library will be required; each application does not keep its own copy.

For most libraries, solidDB provides both a static and a dynamic version on some or all platforms.

In addition, on Windows systems, solidDB provides an import library in some cases. Each import library is associated with a corresponding dynamic link library. Your application will link to the import library. When the application is actually loaded and executed, the operating system will load the corresponding dynamic link library.

Statement cache

Processing of queries is additionally optimized by a built-in "statement cache".

Statement cache is an internal memory storing a few previously prepared SQL statements. The number of cached statements, for a session, can be set by using a client-side `solid.ini` configuration parameter Client.StatementCache. The default value is 6.

The statement cache operates in such a way that the prepare phase is omitted if the prepared statement is in the cache. If a connection is closed, the statement cache is purged.

In JDBC, the statement cache size can be dynamically set by using a non-standard driver property. For more information, see “Non-standard connection properties” on page 66.

2 Using solidDB ODBC API

This section contains solidDB-specific information and usage samples for developing applications that use the ODBC API.

In general, solidDB conforms to the Microsoft ODBC 3.51 standard. solidDB ODBC APIs are defined based on the function prototypes provided by Microsoft. This chapter details those areas where solidDB-specific usage applies and where support for options, data types, and functions differ.

Note: This solidDB Programmer Guide does not contain a full ODBC API reference. For details on developing applications with ODBC API, refer to the *Microsoft Data Access SDK Online ODBC Programmer's Reference*.

solidDB provides two versions of the ODBC driver, one for Unicode and one for ASCII. The Unicode version is a superset of the ASCII version; you may use it with either Unicode or ASCII character sets.

solidDB ODBC Driver 3.51 Features Support

This section provides details about the ODBC Driver 3.51 features support for users who have migrated from a previous version (1.0, 2.0, and 3.0) of the solidDB ODBC Driver to solidDB ODBC Driver 3.51.

The following features are supported in this driver:
- Complete support of descriptors
- All catalog API support
- Unicode support
- Multithread support
- ADO/DAO/RDO/OLE DB support
- Data access through MS Access and MS Query
- Block cursor support

Overview of usage on Windows systems

On the Windows operating systems, the solidDB ODBC Libraries are provided as .DLL files.

The files are named `socw32VV.dll` and `sacw32VV.dll` (where "VV" indicates the version number) for the Unicode and ASCII versions, respectively. For example, the Unicode ODBC driver in version 4.1 is named `socw3241.dll`. To call the functions in one of these .DLL files, you must link to a solidDB import library file. For the solidDBon Microsoft Windows, this import library file is named `solidimpodbcu.lib` (Unicode) or `solidimpodbca.lib` (ASCII). This import library file contains the entry points to the corresponding solidDB ODBC DLL (for example, `socw3241.dll`).

Note: The library files have been produced with Microsoft C++. Other development toolkit manufacturers' linkers may expect different library file formats. In such cases, you might need to use the Import Library utility of the development toolkit to build a library file that is compatible with your linker.

Instructions for usage of solidDB client DLLs (Solid® ODBC Driver files)

There are two alternatives to building application programs that use the solidDB ODBC driver:

1. Using Microsoft ODBC Driver Manager.

   Microsoft ODBC software needs to be installed on all client workstations and a Data Source must be defined using solidDB ODBC Driver. If you use the Driver Manager, then any application that can use the solidDB ODBC driver will also work with any other ODBC compliant engine.

2. Using solidDB ODBC driver directly.

   Connections are opened directly to a server process without using Microsoft ODBC Driver Manager. This usually makes embedded deployment of solidDB easier. However, the application can only use the functions provided by the solidDB library (i.e. the Solid ODBC driver); the application cannot use the ODBC functions that are implemented by the Microsoft ODBC Driver Manager or the Microsoft Cursor library.

solidDB provides some sample programs that can be used either with or without the Microsoft ODBC Driver Manager. These samples are in subdirectories of the "samples" directory that is created when you install the solidDB Development Kit (SDK). Below are brief instructions on how to build and run the provided samples in both of the alternative ways:

1.

   Building the samples to use ODBC Driver Manager.

   a. Create a new application project.
   b. Add the C-source file (e.g. `sqled.c` or `embed.c`) to the project.
   c. Set the solidDB SQL API headers visible to the compiler.
   d. Define SS_WINDOWS for the compiler.
   e. Compile and link.
   f. Make sure that you have installed the solidDB ODBC driver. Also, make sure that the connection string you intend to use is defined as the ODBC data source name.
   g. Run to connect to a listening solidDB server.

2. Building the samples to use solidDB ODBC library directly.

   The necessary changes to the ODBC Driver Manager configuration are listed below.

   a. Add solidDB ODBC driver library file (`solidimpodbcu.lib`) to the project.
   b. Remove ODBC Driver manager libraries `ODBC*.LIB` from the default library list.
   c. Compile and link.
   d. Now it is possible to connect to data sources bypassing ODBC Driver Manager. Make sure that the SQL API DLL `socw32<VV>.dll` (where "VV" indicates the version number) and the solidDB communication DLLs are available. Data Sources may be defined in `solid.ini` or in the ODBC Administration Window.
   e. Run the client to connect to a listening solidDB server.

Calling functions

This section provides information about how programs call functions in the ODBC driver.

Header Files and Function Prototypes

If your program calls functions in the ODBC driver, your program must include the ODBC header files. These files define the ODBC functions, and the data types and constants that are used with ODBC functions. The header files are not solidDB-specific; they are standard header files provided by Microsoft. The solidDB ODBC driver (like any ODBC driver) implements the functions that are specified in these header files.

ASCII and unicode

ODBC drivers come in two "flavors": ASCII and Unicode. The ASCII flavor supports only ASCII character sets. The Unicode flavor supports both the Unicode and the ASCII character sets.

If your program calls only the ASCII flavor of ODBC functions, then you should include the following header files:

- `SQL.H`
- `SQLEXT.H`

If your program calls Unicode ODBC functions, then you should include the following

- `SQLUCODE.H`
- `WCHAR.H` - This file is provided with Microsoft Visual C++ (or Developer Studio).

If your program calls both the ASCII and Unicode flavors of ODBC functions, then you should include just the header files for Unicode. (The Unicode version of the header files also contains definitions for the ASCII functions. In other words, the Unicode headers are a superset of the ASCII headers.)

For details on driver, API, and SQL conformance levels, refer to the Microsoft ODBC API Specification (Part I PDF file), "Introduction to ODBC" available on the solidDB Corporation Web site (http://www.ibm.com/software/data/soliddb).

Using the ODBC Driver Manager

An application may link directly to the solidDB ODBC driver, or the application may link to an ODBC Driver Manager. In this section, we discuss using an ODBC Driver Manager.

On Microsoft Windows, the Driver Manager is required if applications that connect to solidDB use OLE DB or ADO APIs, or you use database tools that require the Driver Manager, such as Microsoft Access, FoxPro, or Crystal Reports. In most other situations, you may link directly to the ODBC driver instead of linking to the Driver Manager.

On Microsoft Windows platforms, Microsoft supplies the Driver Manager, and you link to the Driver Manager import library (`ODBC32.LIB`) to gain access to the Driver

Manager. On other platforms, you can link to another vendor's Driver Manager. For example, on Linux® and Solaris 8, you can use Merant's Driver Manager or iODBC's Driver Manager.

For basic application steps that occur whenever an application calls an ODBC function and details on calling ODBC functions, refer to the Microsoft ODBC API Specification (Part I PDF file), "Introduction to ODBC", available on the solidDB Corporation Web site.

Data types

Appendix E, "Data types," on page 205 provides information about SQL data types that are supported by solidDB. The header files from Microsoft provide information about C-language data types used by your client program. To transfer data between the application program and the database server, you must use appropriate types. For example, on most 32-bit platforms, the C-language "int" data type corresponds to the SQL data type "INT". The C-language "float" data type corresponds to the SQL "REAL" data type. (Note that C "float" does NOT correspond to SQL "FLOAT") For more information about the C-language data types used to transfer data via ODBC calls, you may want to read the appropriate header files: `SQL.H` and `SQLEXT.H` and `SQLUCODE.H` and `WCHAR.H`. Note that `WCHAR.H` contains information regarding the "wide" character format, which corresponds to Unicode.

Scalar functions

Scalar functions return a value for each row. For example, the "absolute value" scalar function takes a numeric column as an argument and returns the absolute value of each value in the column. Scalar functions are invoked with the following ODBC escape sequence:

```
{fn scalar-function}
```

Note: The starting and ending characters are the curly bracket characters, not parentheses.

For a list of scalar functions and a more complete example of their usage, refer to Appendix F, "Scalar functions," on page 251.

solidDBNative Scalar Functions

solidDB provides the following native scalar functions, which cannot be invoked using the ODBC escape sequence. They are:

- CURRENT_CATALOG() - returns a WVARCHAR string that contains the current active catalog name. This name is the same as ODBC scalar function {fn DATABASE()}.
- LOGIN_CATALOG() - returns a WVARCHAR string that contains the login catalog for the connected user (currently the login catalog is the same as the system catalog).
- CURRENT_SCHEMA() - returns a WVARCHAR string that contains the current active schema name.

Function Return Codes

When an application calls a function, the driver executes the function and returns a predefined code. These return codes indicate success, warning, or failure status. The return codes are:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING
- SQL_NEED_DATA

If the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, the application can call SQLError to retrieve additional information about the error.

Connecting to a data source

A data source can be a database server, a flat file, or another source of data.

To access the data source, the application will use the Data Source Name, in SQLConnect() call. The Data Source Name may be given in one of the three following ways: a connect string, a Logical Data Source Name and an empty data source name.

Note: If you are using HotStandby, you have two connectivity types to choose from, the Basic Connectivity and the HotStandby Connectivity. The Basic Connectivity is described below. For further information about the HotStandby Connectivity see the "Using the Transparent Connectivity" topics, in *solidDB High Availability User Guide*.

Using a solidDB Connect String

The solidDB connect string consists of a *communication protocol*, a possible set of *special options*, an optional *host computer name* and a *server name*.

By this combination, the client specifies the server it will establish a connection to. The communication protocol and the server name must match the ones that the server is using in its network listening name. In addition, most protocols need a specified host computer name if the client and server are running on different machines. All components of the client's network name are case insensitive.

Note: In HSB configurations, the connect string may take a more general form of TC info (Transparent Connectivity Info). For more information, see the*solidDB High Availability User Guide*.

The same format of the connect string applies to both the connect configuration parameters in the `solid.ini` file and data source names used in ODBC applications.

The format of a connect string is the following:

```
protocol_name [options] [server_name] [port_number]
```

where options may be any number of:

Table 4. Connect String Options

Option	Meaning
-z	Data compression is enabled for this connection
-c *milliseconds*	Login timeout is specified (the default is operating-system-specific). A login request fails after the specified time has elapsed. **Note:** applies for the TCP protocol only.
-r *milliseconds*	Connection (or read) timeout is specified (the default is 60 s). A network request fails when no response is received during the time specified. The value 0 sets the timeout to infinite. **Note:** applies for the TCP protocol only..

Examples

```
tcp localhost 1315
tcp 1315
tcp -z -c1000 1315
nmpipe host22 SOLIDDB
```

Using a logical data source name

If the data source name is not a valid solidDB connect string, the driver assumes it is a Logical Data Source Name.

solidDB clients support Logical Data Source Names. These names can be used for giving a database a descriptive name. This name can be mapped to a data source in three ways:

1. Using the parameter settings in the application's `solid.ini` file.
2. Using the Microsoft Windows operating system's registry settings.
3. Using settings in a `solid.ini` file located in the Windows directory.

This feature is available on all supported platforms. However, on non-Windows platforms, only the first method is available.

When you call the SQLConnect() in Windows, the solidDB ODBC Driver will check all logical data sources in the ODBC registry to find a mapping between the Logical Data Source Name and a valid solidDB ODBC Driver connection string. The time consumed for this operation is proportional to the amount of defined data sources. You can expect connection times as follows:

- With only few (1 to 5) data sources, the connection time will be approximately 5 ms.
- With 1000 data sources, the connection time will be approximately 200 ms.

A solidDB client attempts to open the file `solid.ini` first from the directory set by the SOLIDDIR environment variable. If the file is not found from the path specified by this variable or if the variable is not set, an attempt is made to open the file from the current working directory.

To define a Logical Data Source Name using the `solid.ini` file, you need to create a `solid.ini` file containing the section [Data Sources]. In that section, you need to enter the 'logical name' and 'network name' pairs that you want to define. The syntax of the parameters is the following:

```
[Data Sources]
logical_name = connect_string, Description
```

In the description field, you may enter comments on the purpose of this logical name.

Note: solidDB ODBC Driver will check for existence of the `solid.ini` file every time a connection is attempted. If the file system is particularly slow, for example because the working directory is mapped to a network drive, this can have a measurable performance impact. However, if the `solid.ini` file contains mappings between the Data Source Name and the network name in the [Data Sources] section, the driver does not try to access the registry for the mapping.

For example, assume you want to define a logical name for the application *My_application* and the database that you want to connect is located in a UNIX server using TCP/IP. Then you should include the following lines in the `solid.ini` file, which you need to place in the working directory of your application:

```
[Data Sources]
My_application = tcpip irix 1313, Sample data source
```

When your application now calls the Data Source 'My_application', the solidDB client maps this to a call to 'tcpip irix 1313'.

On Windows platforms, the registry is typically used to map Data Sources. To setup the registry with a GUI interface, use the Windows Administrative Control Panel "Data Sources (ODBC)".

In detail, the following are the rules for mapping ODBC data sources in the registry:

The entry is searched from the path `software\odbc\odbc.ini`

1. first, under the root HKEY_CURRENT_USER and if not found,
2. under the root HKEY_LOCAL_MACHINE.

The order of resolving a Data Source name in Microsoft Windows systems is the following:

1. Look for the Data Source Name from the `solid.ini` file in the current working directory, under the section [Data Source]
2. Look for the Data Source Name from the following registry path:
   HKEY_CURRENT_USER\software\odbc\odbc.ini\DSN
3. Look for the Data Source Name from the following registry path
   HKEY_LOCAL_MACHINE\software\odbc\odbc.ini\DSN

Note: The solidDB ODBC Driver checks all logical data sources in the ODBC registry to find a mapping between a logical data source name and a valid solidDB ODBC Driver connect string. The time consumed for this operation is proportional to the amount of defined data sources. An SQLConnect() connection time with 1000 data sources is approximately 200 ms.

Applications that bypass the Driver Manager to access data from solidDB databases by directly linking with the driver, must connect to the server using a valid connect string. If the data source name is not a valid solidDB connect string, all solidDB client applications search for a valid data source name in:

1. the `solid.ini` file

2. the ODBC.INI or registry.

Empty data source name

When an application uses the ODBC API directly and calls SQLConnect() without specifying a solidDB server network name (by giving an empty string), it is read from the parameter Connect in the [Com] section of the client application's solid.ini file.

The solid.ini file must reside in the current working directory of the application or in a path specified by the SOLIDDIR environment variable.

The following connect line in the solid.ini of the application workstation will connect an application (client) using the TCP/IP protocol to a solidDB server running on a host computer named 'spiff' and listening with the name (port number in this case) '1313'.

```
[Com]
Connect = tcpip spiff 1313
```

If the Connect parameter is not found in the solid.ini configuration file, then the client uses the environment-dependent default instead. The defaults for the Listen and Connect parameters are selected so that the application (client) will always connect to a local solidDB server listening with a default network name. So local communication (inside one machine) does not necessarily need a configuration file for establishing a connection.

Configuring the solidDB ODBC Data Source for Windows

To configure an ODBC data source for Windows platforms, you need to perform the steps described in this section.

Before you begin

To be able to configure solidDB ODBC data sources, the solidDB ODBC Driver must be installed. The driver installation is discussed in "Installing and configuring ODBC software" on page 47.

Procedure

1. Invoke **Data Sources (ODBC)** from **Control Panel** > **Administrative Tools**
2. Open the **User DSN** tab.
3. Click the **Add...** button.
4. Select the solidDB ODBC Driver (ANSI or UNICODE according to your database requirements).
5. Enter the Data Source configuration in the solidDB ODBC Driver Setup box as shown in the following example.

   Note: The **NetworkName** entry should be compliant with the database server listen addresses defined in solid.ini. The network name follows the connection string format presented in "Format of the connect string" on page 2.

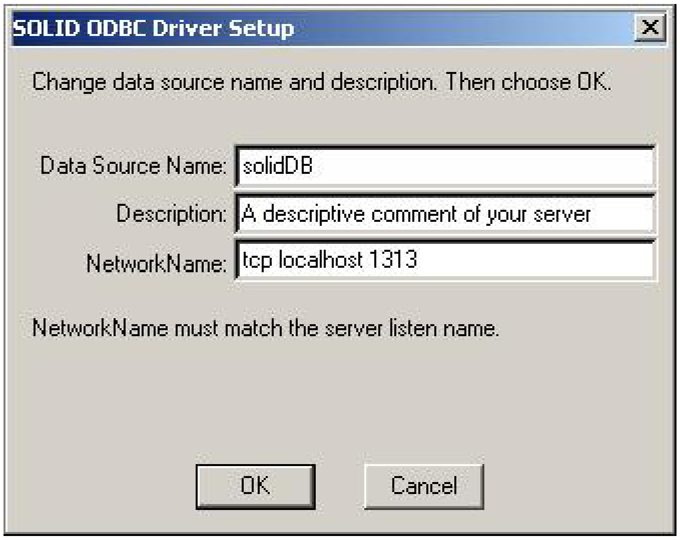

Figure 1. ODBC driver setup

Retrieving user login information

This section describes how the Driver Manager retrieves login information.

If the application calls SQLDriverConnect() and requests that the user be prompted for information, the Driver Manager displays a dialog box similar to the following example:

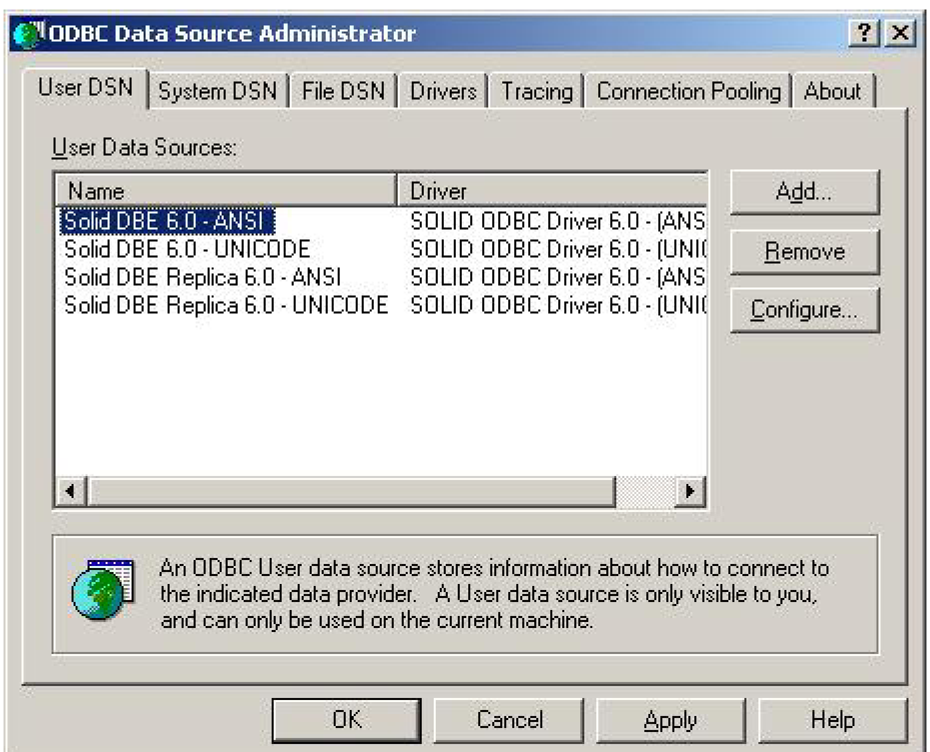

Figure 2. ODBC data source administrator

On request from the application, the driver retrieves login information by displaying a dialog box.

ODBC handle validation

You can control ODBC handle validation with the client-side **Com.ODBCHandleValidation** parameter or dynamically with the non-standard ODBC attribute `SQL_ATTR_HANDLE_VALIDATION`. For performance reasons, ODBC handle validation in solidDB is switched off by default.

For example, in Windows environments with ODBC driver manager, the driver manager performs the handle validation and the solidDB ODBC driver does not need to repeat the same validation procedures. Also, a carefully written ODBC application does not normally cause invalid handles to be used; in that case, the handle validation in the ODBC driver is not needed. In both cases, the applications may benefit from performance improvements when skipping the handle validation in the driver.

To switch ODBC handle validation on or off:

- Set the client-side **Com.ODBCHandleValidation** to 'Yes' or 'No'. Default is 'No'.

  ```
  [Com]
  ODBCHandleValidation=Yes
  ```

  or
- Set the non-standard environment attribute `SQL_ATTR_HANDLE_VALIDATION` to 1 (on) or 0 (off). Default is 0.
  - To switch handle validation on:

    ```
    SQLSetEnvAttr(henv, SQL_ATTR_HANDLE_VALIDATION, (SQLPOINTER)1, 0);
    ```
  - To switch handle validation off:

    ```
    SQLSetEnvAttr(henv, SQL_ATTR_HANDLE_VALIDATION, (SQLPOINTER)0, 0);
    ```

  Important: The `SQL_ATTR_HANDLE_VALIDATION` attribute must be set after creating the environment handle but before any other handle is created. The `SQL_ATTR_HANDLE_VALIDATION` attribute is global; when set, it affects all the solidDB ODBC handles initiated by the application. This ensures consistency by preventing the application from allocating both validated and non-validated handles.

When the handle validation is switched on, any ODBC function may fail with the standard return value SQL_INVALID_HANDLE.

If handle validation is turned off and invalid handle is used by the application, the ODBC driver behavior is unpredictable and most likely causes the application to crash.

Related concepts

"solidDB Extensions for ODBC API" on page 27
The following functions and connection attributes are solidDB-specific extensions to ODBC API.

Related reference

"Descriptions of client-side configuration parameters" on page 278
This topic describes the three section of the`solid.ini` file. The sections are the communication, data sources, and client.

Executing transactions

This section provides information on how transactions are committed.

In *auto-commit* mode, each SQL statement is a complete transaction, which is automatically committed when the statement finishes executing. Refer to the important notes in the *Committing Read-Only Transactions* section on committing read-only SELECTs.

In *manual-commit* mode, a transaction consists of one or more statements. In manual-commit mode, when an application submits a SQL statement and no transaction is open, the driver implicitly begins a transaction. The transaction remains open until the application commits or rolls back the transaction with SQLEndTran.

Committing Read-Only Transactions

Important:

- When the isolation level is other than READ COMMITED, even read-only statements (e.g. SELECT) must be committed. Furthermore, the user must commit SELECT statements even if the server is in autocommit mode. Failure to commit statements can reduce performance or cause the server to run out of memory. This is explained in more detail below.
- If the isolation level is READ COMMITTED, read-only statements need not be committed. In that case, the explanation below does not apply.

Even a read-only statement must be committed. The reason for this is that solidDB saves the 'read-level' of each transaction and until that transaction commits, all subsequent transactions from other connections are also maintained in memory. (This behavior is part of the row versioning performed by the Bonsai Tree technology. See *solidDB Administration Guide* for more details about the Bonsai Tree.) If a transaction is not committed, the server will need more and more memory as other transactions accumulate; this will reduce performance, and eventually the server may run out of available memory. For more details, read the Performance Tuning chapter in *solidDB Administration Guide*.

SELECT and autocommit

Using autocommit mode does not ensure that SELECT statements are committed. The server cannot automatically commit SELECTs because SELECTs do not execute as a single statement. Each SELECT involves opening a cursor, fetching rows, and then closing the cursor.

There are two possible ways that the server could automatically commit when fetching multiple rows: the server could commit after the final fetch, or the server could commit after each individual fetch. Unfortunately, neither of these is practical, and therefore the server cannot commit the SELECT statement even in autocommit mode.

The server cannot automatically commit after the final fetch because the server does not know which fetch is the final fetch — the server does not know how many rows the user will fetch. (Until the user closes the cursor, the server does not know that the user is done fetching.)

It is not practical to commit after each individual fetch because each transaction should see the data as it was at the time that the transaction started, and therefore if each fetch is in a different transaction then the data can be from a different "snapshot" of the database. Putting each fetch in a different transaction would also make REPEATABLE READ and SERIALIZABLE transaction isolation levels confusing or meaningless for the cursor, even though the cursor is for a single SELECT statement.

To commit the SELECT statement, the user may:
- Execute an explicit COMMIT WORK statement.
- Execute a statement to which autocommit does apply (i.e. a statement other than SELECT).
- If the cursor is the only open cursor, then the user may commit by explicitly closing the cursor (the server automatically commits when a cursor is closed and there are no other open cursors (and the server is in autocommit mode). This is part of why we recommend that you explicitly close every cursor as soon as you are done with it.

Note: To ensure that the data in the cursor is consistent and recent, the server actually does an automatic commit immediately prior to opening the cursor (if autocommit is on). The server then immediately starts a new transaction to contain the subsequent FETCH statement(s).This new transaction, like any other transaction, must be committed (or rolled back).

Summary

All statements must be committed, even if they are read-only statements, if an isolation level other than READ COMMITTED is used.

In most cases when you are doing SELECT statements in autocommit mode, you should explicitly close each cursor as soon as you are done with it and then explicitly COMMIT, even though you are in autocommit mode.

Retrieving information about the data source's catalog

This section describes functions (known as *catalog functions*) that return information about a data source's catalog.
- SQLTables returns the names of tables stored in a data source.
- SQLTablePrivileges returns the privileges associated with one or more tables.
- SQLColumns returns the names of columns in one or more tables.
- SQLColumnPrivileges returns the privileges associated with each column in a single table.
- SQLPrimaryKeys returns the names of columns that comprise the primary key of a single table.

- SQLForeignKeys returns the names of columns in a single table that are foreign keys. It also returns the names of columns in other tables that refer to the primary key of the specified table.
- SQLSpecialColumns returns information about the optimal set of columns that uniquely identify a row in a single table or the columns in that table that are automatically updated when any value in the row is updated by a transaction.
- SQLStatistics returns statistics about a single table and the indexes associated with that table.
- SQLProcedures returns the names of procedures stored in a data source.
- SQLProcedureColumns returns a list of the input and output parameters, as well as the names of columns in the resultset, for one or more procedures.

Each function returns the information as a resultset. An application retrieves these results by calling SQLBindCol() and SQLFetch().

Executing Functions Asynchronously

Note: ODBC drivers in all solidDB products do not support asynchronous execution.

Using ODBC extensions to SQL

ODBC defines extensions to SQL, which are common to most database management systems.

For details on SQL extensions, refer to "Escape Sequences in ODBC" in the Microsoft ODBC API Specification (Part I PDF file that is available on the IBM Corporation Web site) which contains the introductory part of the Microsoft ODBC *Programmer's Reference*.

Included in the ODBC extensions to SQL are:
- Procedures
- Hints

Details on solidDB usage for these extensions are described in the following sections.

Procedures

Stored procedures are procedural program code containing one or more SQL statements and program logic.

Stored procedures are stored in the database and executed with one call from the application or another stored procedure. Read the description of Stored Procedures in *solidDB SQL Guide* for a full description of solidDB stored procedures.

An application can call a procedure in place of a SQL statement. The escape clause ODBC uses for calling a procedure is:

```
{call procedure-name [([parameter][,[parameter]]...)]}
```

where procedure-name specifies the name of a procedure stored on the data source and parameter specifies a procedure parameter.

Note: The ODBC standard shows the escape clause as:

```
{[?=] call procedure-name [([parameter][,[parameter]]...)]}
```

However, solidDB does not support the optional "?=" part of the syntax. (Earlier versions of *solidDB Programmer Guide* stated that solidDB supported the "?=" syntax, but were incorrect.)

A procedure can have zero or more parameters. For input and input/output parameters, *parameter* can be a literal or a parameter marker. Because some data sources do not accept literal parameter values, be sure that interoperable applications use parameter markers. For output parameters, *parameter* must be a parameter marker. If a procedure call includes parameter markers, the application must bind each marker by calling SQLBindParameter() prior to calling the procedure.

Procedure calls do not require input and input/output parameters. Note the following rules:

- A procedure called with parentheses but with parameters omitted, such as {call *procedure_name()*} may cause the procedure to fail.
- A procedure called without parentheses, such as {call *procedure_name*}, returns no parameter values.
- Input parameters may be omitted. Omitted input or input/output parameters cause the driver to instruct the data source to use the default value of the parameter. As an option, a parameter's default value can be set using the value of the length/indicator buffer bound to the parameter to SQL_DEFAULT_PARAM.
- When a parameter is omitted, the comma delimiting it from other parameters must be present.
- Omitted input/output parameters or literal parameter values cause the driver to discard the output value.
- Omitted parameter markers for a procedure's return value cause the driver to discard the return value.
- If an application specifies a return value parameter for a procedure that does not return a value, the driver sets the value of the length/indicator buffer bound to the parameter to SQL_NULL_DATA.

To determine if a data source supports procedures, an application calls SQLGetInfo() with the SQL_PROCEDURES information type. For more information about procedures, read the description of Stored Procedures in *solidDB SQL Guide*.

Hints

Within a query, Optimizer directives or *hints* can be specified to determine the query execution plan that is used.

Hints are detected through a pseudo comment syntax from SQL-92. solidDB provides its own extensions to hints:

```
--(* vendor (Solid), product (Engine), option(hint)
--hint
-- *)--
hint :=
    [MERGE JOIN |
    TRIPLE MERGE JOIN |
    LOOP JOIN |
    JOIN ORDER FIXED |
    INTERNAL SORT |
    EXTERNAL SORT |
    INDEX [REVERSE] table_name.index_name |
    PRIMARY KEY [REVERSE] table_name |
    FULL SCAN table_name |
```

```
[NO] SORT BEFORE GROUP BY |
UNION FOR OR |
OR FOR OR |
LOOP FOR OR]
```

The pseudo comment prefix is followed by identifying information. Vendor is specified as *Solid*, product as *Engine*, and the option, which is the pseudo comment class name, as a valid hint.

The terminator may be on its own line, or it may be at the end of the last line of the hint. For example, either of the following is acceptable:

```
--(* vendor (Solid), product (Engine), option(hint)
--hint
-- *)--
```

or

```
--(* vendor (Solid), product (Engine), option(hint)
--hint *)--
```

Note that spacing is sensitive. In the pseudo comment prefix --(* and postfix *)-- , there can be no space between the parenthesis and the asterisk. There must be a space prior to the *)-- terminator, i.e. prior to the asterisk (see the examples above). No space is required prior to the opening parenthesis in --(* . The terminator *)-- cannot be on a line by itself without being after the comment delimiter --.

A hint always follows the SELECT, UPDATE, or DELETE keyword that it applies to.

Note: Hints are not allowed after the INSERT keyword.

Each subselect requires its own hint; for example, the following are valid uses of hints syntax:

```
INSERT INTO ... SELECT hint FROM ...
UPDATE hint TABLE ... WHERE column = (SELECT hint ... FROM ...)
DELETE hint TABLE ... WHERE column = (SELECT hint ... FROM ...)
```

Hints Example 1

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--MERGE JOIN
--JOIN ORDER FIXED
-- *)--
col1, col2
FROM TAB1 A, TAB2 B;
WHERE A.INTF = B.INTF;
```

Hints Example 2

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--INDEX TAB1.INDEX1
--INDEX TAB1.INDEX1 FULL SCAN TAB2
-- *)--
*
FROM TAB1, TAB2
WHERE TAB1.INTF = TAB2.INTF;
```

Hint uses a specific semantic, corresponding to a specific behavior. Following is a list of solidDB-supported hints:

Table 5. solidDB-supported hints

Hint	Definition
MERGE JOIN	Directs the Optimizer to choose the merge join access plan in a select query for all tables listed in the FROM clause. The MERGE JOIN option is used when two tables are approximately equal in size and the data is distributed equally. It is faster than a LOOP JOIN when an equal amount of rows are joined. For joining data, MERGE JOIN supports a maximum of three tables. The joining table is ordered by joining columns and combining the results of the columns. You can use this hint when the data is sorted by a join key and the nested loop join performance is not adequate. The Optimizer selects the merge join only where there is an equal predicate between tables (e.g. "table1.col1 = table2.col1"). Otherwise, the Optimizer selects LOOP JOIN even if the MERGE JOIN hint is specified. Note that when data is not sorted before performing the merge operation, the solidDB query executor sorts the data. Keep in mind that the merge join with a sort is more resource intensive than the merge join without the sort.
TRIPLE MERGE JOIN	TRIPLE MERGE JOIN is a variant of MERGE JOIN. It has three table sources which are merged on equal basis instead of the two in MERGE JOIN. The TRIPLE MERGE JOIN hint instructs the SQL interpreter to use the triple merge join algorithm whenever possible. The triple merge join algorithm can only be used in situations where in all three table sources there is one single field that should be equal in all the resulting rows after evaluating the WHERE condition.
LOOP JOIN	Directs the Optimizer to pick the nested loop join in a select query for all tables listed in the FROM clause. By default, the Optimizer does not pick the nested loop join. The LOOP JOIN loops through both inner and outer tables to find matches between columns in the inner and outer tables. For better performance, the joining columns should be indexed. Using the loop join when tables are small and fit in memory may offer greater efficiency than using other join algorithms.
JOIN ORDER FIXED	Specifies that the Optimizer use tables in a join in the order listed in the FROM clause of the query. This means that the Optimizer does not attempt to rearrange the join order and does not try to find alternate access paths to complete the join. We recommend that you "test" the hint by running the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query.

Table 5. solidDB-supported hints (continued)

Hint	Definition
INTERNAL SORT	Specifies that the query executor use the internal sorter. Use this hint if the expected resultset is small (hundreds of rows as opposed to thousands of rows); for example, if you are performing some aggregates, ORDER BY with small resultsets, or GROUP BY with small resultsets, etc. This hint avoids the use of the more expensive external sorter.

EXTERNAL SORT	Specifies that the query executor use the external sorter. Use this hint when the expected resultset is large and does not fit in memory; for example, if the expected resultset has thousands of rows.
	
	In addition, specify the SORT working directory in the `solid.ini` before using the external sort hint. If a working directory is not specified, you will receive a run-time error. The working directory is specified in the [sorter] section of the `solid.ini` configuration file. For example:
	`[sorter]`
	`TmpDir_1=c:\soldb\temp1`

INDEX [REVERSE] *table_name.index_name*	Forces a given index scan for a given table. In this case, the Optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query. We recommend that you "test" the hint by running the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query. The optional keyword REVERSE returns the rows in the reverse order. In this case, the query executor begins with the last page of the index and starts returning the rows in the descending (reverse) key order of the index. Note that in *tablename.indexname*, the tablename is a fully qualified table name which can include the *catalogname* and *schemaname*.
PRIMARY KEY [REVERSE] *table_name*	Forces a primary key scan for a given table. The optional keyword REVERSE returns the rows in the reverse order. If the primary key is not available for the given table, then you will receive a run-time error.
FULL SCAN *table_name*	Forces a table scan for a given table. In this case, the optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query. Before using this hint, it is recommended that you "test" the hint by running the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query.

Table 5. solidDB-supported hints (continued)

Hint	Definition
[NO] SORT BEFORE GROUP BY	Indicates whether the SORT operation occurs before the resultset is grouped by the GROUP BY columns. If the grouped items are few (hundreds of rows) then use NO SORT BEFORE. On the other hand, if the grouped items are large (thousands of rows), then use SORT BEFORE.

UNION FOR OR	The UNION FOR OR hint instructs the SQL interpreter to replace an OR condition of style `A = 1 OR A = 2` with a construction of the following type:
	`SELECT ... WHERE A = 1`
	`UNION ALL`
	`SELECT ... WHERE A = 2`
	
	In most cases the SQL interpreter performs the replacement automatically; the UNION FOR OR hint ensures the UNION-type execution is used always.
	Note: Conditions of type `A = 1 OR B = 2` can also be handled, but this may be problematic since the conditions are not mutually exclusive. Because of this, the construction for `A = 1 OR B = 2` is the following:
	`SELECT ... WHERE A = 1`
	`UNION ALL`
	`SELECT ... WHERE B = 2 AND UtoT NOT (A = 1)`
	
	where `UtoT` stands for UNKNOWN TO TRUE.
	
	The `UtoT` operator is needed for handling cases with NULL values. Without the `UtoT` operator, a row which has values `A = NULL` and `B = 2` would not appear correctly in the UNION variant.

OR FOR OR	The OR FOR OR hint is the opposite for UNION FOR OR. It prevents the interpreter from using the UNION-type solution.
LOOP FOR OR	The LOOP FOR OR hint is an alternative query execution plan that falls between UNION FOR OR and OR FOR OR. With LOOP FOR OR the OR values are passed individually to the data table level, but conditions like `A = 1 OR B = 2` cannot be handled (see description of UNION FOR OR for details on `A = 1 OR B = 2`).

Additional ODBC extension functions

ODBC provides the following functions related to SQL statements.

Refer to the Microsoft ODBC API Specification (Part II PDF file that is available on the IBM Corporation Web site) for more information about these functions.

Table 6. Additional ODBC Extension Functions

Function	Description
SQLDescribeParam	Retrieves information about prepared parameters.

Table 6. Additional ODBC Extension Functions (continued)

Function	Description
SQLNumParams	Retrieves the number of parameters in a SQL statement.
SQLSetStmtAttr SQLSetConnectAttr SQLGetStmtAttr	These functions set or retrieve statement options, such as asynchronous processing, orientation for binding rowsets, maximum amount of variable length data to return, maximum number of resultset rows to return, and query timeout value. Note that SQLSetConnectAttr sets options for all statements in a connection.

solidDB Extensions for ODBC API

The following functions and connection attributes are solidDB-specific extensions to ODBC API.

Non-standard ODBC functions

Table 7. solidDB-specific ODBC functions to ODBC API

Function	Description
SQLFetchPrev	This function is the same as the ODBC function SQLFetch, but for fetching previous record.
SQLSetParamValue	This function sets the value of a parameter marker in the SQL statement specified in SQLPrepare. Parameter markers are numbered sequentially from left-to-right, starting with one, and may be set in any order.
SQLGetCol	This function is the same as the ODBC function SQLGetData.
SQLGetAnyData	This function is the same as the ODBC function SQLGetData.

Non-standard ODBC attributes

The following connection attributes are specific to solidDB.

Note: If the attribute is marked as OUT, it is a read-only attribute and cannot be set through the ODBC interface.

- SQL_ATTR_TF_LEVEL

  OUT: integer (TF level: 0=NONE, 1=CONNECTION, 3=SESSION)

  The failure transparency level.
- SQL_ATTR_TC_PRIMARY

  OUT: string, Primary server connection string

  There is always a value indicating the current Primary server.
- SQL_ATTR_TC_SECONDARY

  OUT: string, Secondary server connection string

  The value indicates the assigned workload server if:

  1. PA=READ_MOSTLY, and
  2. the Secondary is the designated workload server.

Otherwise, the returned string is empty.

- SQL_ATTR_TF_WAITING

  OUT: string, Secondary server connection string.

  The value indicates the assigned watchdog (waiting) connection. Waiting connection is used by ODBC driver internally to detect possible looses (crashes, unavailability) of the primary server faster. The string is empty if the connection is not a TC connection.
- SQL_ATTR_PA_LEVEL

  OUT: integer (Preferred Access level: 0=WRITE_MOSTLY, 1=READ_MOSTLY)

  The attribute indicates whether the load balancing is used or not.
- SQL_ATTR_TC_WORKLOAD_CONNECTION

  OUT: string, server name of the workload connection

  The current workload connection server; if queried before the Commit, the value indicates the server the transaction will be committed on. It may be queried as the statement attribute as well. In that case, it indicates the server the next statement will be executed on.
- SQL_ATTR_LOGIN_TIMEOUT_MS

  IN/OUT: integer, login timeout in milliseconds

  Note: There is also a standard attribute SQL_ATTR_LOGIN_TIMEOUT that can be used to set the timeout in seconds.
- SQL_ATTR_CONNECTION_TIMEOUT_MS

  IN/OUT: integer, connection timeout in milliseconds

  Note: There is also a standard attribute SQL_ATTR_CONNECTION_TIMEOUT that can be used to set the timeout in seconds.
- SQL_ATTR_QUERY_TIMEOUT_MS

  IN/OUT: integer, query timeout in milliseconds

  Note: There is also a standard attribute SQL_ATTR_QUERY_TIMEOUT that can be used to set the timeout in seconds.
- SQL_ATTR_IDLE_TIMEOUT

  IN/OUT: integer, connection idle timeout in minutes

  Indicates the connection specific idle timeout to be used by the server. If there is no activity on the connection for specified time period, the server automatically shuts down the connection, effectively throwing out the user.

  Special semantics:

  – -1 (default) - the connection timeout is equal to the server default

  – 0 - no idle timeout, connection is never closed

  This property value can be set only before executing SQLConnect().
- SQL_ATTR_HANDLE_VALIDATION (environment handle attribute)

  IN/OUT: integer, turns ODBC standard handle validation on (1) or off (0).

  Default is 0.

  This attribute is global, meaning that once it is set, it affects all the solidDB ODBC connections initiated by the application. In certain systems, for example in Windows with ODBC driver manager involved. the driver manager performs the handle validation and the solidDB ODBC driver does not have to repeat the same validation procedures by itself. Also, a carefully written ODBC application normally does not cause invalid handles to be used; in that case, the handle validation in the ODBC driver is not needed. In both cases, the applications may benefit from performance improvements when skipping the handle validation in the driver. In the case the handle validation is turned off, and invalid handle is used by the application, the ODBC driver behavior is unpredictable and, most likely, it causes the application to crash.

- SQL_ATTR_SET_CONNECTION_DEAD

  IN/OUT: integer, should be set to 1 when needed

  When this attribute is set on a connection, it causes the driver to abort the connection forcibly, without a disconnecting handshake with the server. After the attribute is set to 1, the connection becomes unusable.

Using cursors

The ODBC Driver uses a *cursor* concept to keep track of its position in the resultset, that is, in the data rows retrieved from the database. A cursor is used for tracking and indicating the current position, as the cursor on a computer screen indicates current position.

Each time an application calls SQLFetch, the driver moves the cursor to the next row and returns that row. An application can also call SQLFetchScroll or SQLExtendedFetch (ODBC 2.x), which fetches more than one row with a single fetch or call into the application buffer. This is known as "block cursor" support. Note that the actual number of rows fetched depends upon the rowset size specified by the application.

An application can call SQLSetPos to position a cursor within a fetched block of data using the SQL_POSITION option. This allows an application to refresh data in the rowset. SQLSetPos is also called to update data with the SQL_UPDATE option or delete data in the resultset with the SQL_DELETE option.

The cursor supported by the core ODBC functions only scrolls forward, one row at a time. (To re-retrieve a row of data that it has already retrieved from the resultset, the application must close the cursor by calling SQLFreeStmt with the SQL_CLOSE option, re-execute the SELECT statement, and fetch rows with SQLFetch, SQLFetch Scroll, or SQLExtendedFetch (ODBC 2.x) until the target row is retrieved.) If you need the ability to scroll backward as well as forward, please use block cursors.

Assigning storage for rowsets (binding)

In addition to binding individual rows of data, an application can call SQLBindCol to assign storage for a *rowset* (one or more rows of data). By default, rowsets are bound in column-wise fashion. They can also be bound in row-wise fashion.

To specify how many rows of data are in a rowset, an application calls SQLSetStmtAttr with the SQL_ROWSET_SIZE option.

Column-wise binding

To assign storage for column-wise bound results, an application performs the following steps for each column to be bound:

1. Allocates an array of data storage buffers. The array has as many elements as there are rows in the rowset.
2. Allocates an array of storage buffers to hold the number of bytes available to return for each data value. The array has as many elements as there are rows in the rowset.

3. Calls SQLBindCol and specifies the address of the data array, the size of one element of the data array, the address of the number-of-bytes array, and the type to which the data will be converted. When data is retrieved, the driver will use the array element size to determine where to store successive rows of data in the array.

Row-wise binding

To assign storage for row-wise bound results, an application performs the following steps:

1. Declares a structure that can hold a single row of retrieved data and the associated data lengths. (For each column to be bound, the structure contains one field to contain data and one field to contain the number of bytes of data available to return.)
2. Allocates an array of these structures. This array has as many elements as there are rows in the rowset.
3. Calls SQLBindCol for each column to be bound. In each call, the application specifies the address of the column's data field in the first array element, the size of the data field, the address of the column's number-of-bytes field in the first array element, and the type to which the data will be converted.
4. Calls SQLSetStmtAttr with the SQL_BIND_TYPE option and specifies the size of the structure. When the data is retrieved, the driver will use the structure size to determine where to store successive rows of data in the array.

Cursor support

Applications require different means to sense changes in the tables underlying a resultset. Various cursor models are designed to meet these needs, each of which requires different sensitivities to changes in the tables underlying the resultset.

For example, when balancing financial data, an accountant needs data that appears static; it is impossible to balance books when the data is continually changing. When selling concert tickets, a clerk needs up-to-the minute, or dynamic, data on which tickets are still available.

solidDB cursors which are set with SQLSetStmtAttr as "dynamic" closely resemble static cursors, with some dynamic behavior. solidDB dynamic cursor behavior is static in the sense that changes made to the resultset by other users are not visible to the user, as opposed to ODBC dynamic cursors in which changes are visible to the user.

In solidDB, as long as the cursor scrolls forward from block to block and never scrolls backward or the cursors move back and forth within the same block after an update is done, then the user gets the dynamic cursor behavior. This means that all changes are visible. Note, however that this behavior is affected by the solidDB AUTOCOMMIT mode setting. For details, read “Cursors and autocommit” on page 31. For an example of cursor behavior when using SQLSetPos, read “Cursors and positioned operations” on page 33.

Another characteristic of solidDB's cursor behavior is that transactions are able to view their own data changes (with some limitations), but cannot view the changes made by other transactions that overlap in time. (For more details about the limitations on users seeing their own data changes, refer to “Cursors and positioned operations” on page 33). For example, once Transaction_A starts, it will not see any changes made by any other transaction that did not commit work before Transaction_A started. The conditions in solidDB that cause a user's own changes to be invisible to that user are:

- In a SELECT statement when an ORDER BY clause or a GROUP BY clause is used, solidDB caches the resultset, which causes the user's own change to be invisible to the user.
- In applications written using ADO or OLE DB, solidDB cursors are more like dynamic ODBC cursors to enable functions such as a rowset update.

Specifying the cursor type

To specify the cursor type, an application calls SQLSetStmtAttr with the SQL_CURSOR_TYPE option. The application can specify a cursor that only scrolls forward, a static cursor, or a dynamic cursor.

Unless the cursor is a forward-only cursor, an application calls SQLExtendedFetch (ODBC 2.x) or SQLFetchScroll (ODBC 3.x) to scroll the cursor backwards or forwards.

Cursor support

This section describes the cursor type supported by solidDB.

Three types of cursors are defined in ODBC 3.51:

- Driver Manager supported cursors
- Server supported cursors
- Driver supported cursors

solidDB cursors are server supported cursors.

Cursors and autocommit

This section provides information on cursors and autocommit.

For solidDB-specific information on cursors and autocommit, read "Committing Read-Only Transactions" on page 19.

There are also some limitations in using the solidDB Autocommit mode if your application uses block cursors and positioned updates and deletes. For a brief description of these cursor features, read "Using cursors" on page 29.

When using block cursors and positioned updates and deletes, you must:

- In the application, set commit mode to SQL_AUTOCOMMIT_OFF.
- Commit changes in the application only when all the fetch and positioned operations are done.
- In between positioned operations, be sure not to commit the changes.

Attention:

If the application uses commit mode as SQL_AUTOCOMMIT_ON or commits the changes before it is done with all the positioned operation, then the application may experience unpredictable behavior while browsing through the resultset. Read the section below for details.

Positioned Cursor Operations and SQL_AUTOCOMMIT_ON

The solidDB ODBC Driver keeps a row number/counter for every row in the rowset, which is the data rows retrieved from the database. When an application has the commit mode set to SQL_AUTOCOMMIT_ON and then executes a positioned update or a delete on a row in the rowset, the row is immediately updated in the database. Depending on the new value of the row, the row may be moved from its original position in the resultset. Since the updated row has now moved and its new position is unpredictable (since it is totally dependent on the new value), the driver loses the counter for this row.

In addition, the counter for all other rows in the rowset may also become invalid because of a change in position of the updated row. Hence the application may see incorrect behavior when it does the next fetch or SQLSetPos operation.

Following is an example that explains this limitation.

Assume an application performs the following steps:
1. Sets the commit mode to SQL_AUTOCOMMIT_ON.
2. Sets the rowset size to 5.
3. Executes a query to generate a resultset containing n rows.
4. Fetches the first rowset of 5 rows with SQLFetchScroll.

   A sample resultset is shown below. In the sample, the resultset has only 1 column (defined as varchar(32)). In the table below, the first column shows the row number maintained by the driver internally. The second column shows the actual row values.

Table 8. A Sample Resultset

Row Counter Stored Internally by the Driver	Row Value
1	Antony
2	Ben
3	Charlie
4	David
5	Edgar

Assume now that the application calls SQLSetPos to update the third row with a new value of Gerard. To perform the update, the new row value is moved and positioned as shown below:

Table 9. A sample resultset

Row Counter Stored Internally by the Driver	Row Value
1	Antony
2	Ben

Table 9. A sample resultset (continued)

Row Counter Stored Internally by the Driver	Row Value
Empty row	
4	David
5	Edgar
New row	Gerard

Now the row counter for "David" becomes 3 and not 4, while the counter for "Edgar" becomes 4 and not 5. Since some row counters are now invalid, they will give wrong results when used by the driver to do relative or absolute positioning of the cursor.

If the commit mode had been set to SQL_AUTOCOMMIT_OFF, the database is not updated until the SQLEndTran function is called to commit the changes.

For solidDB-specific information on cursors and autocommit, read "Committing Read-Only Transactions" on page 19.

Cursors and positioned operations

When an application is performing positioned operations (such as updates and deletes when calling SQLSetPos), there are limitations in resultset visibility.

Case 1 illustrates cursor behavior when using SQLSetPos. In Case 1, the cursor scrolls back and forth within the same block after the update is applied.

Although Case 1 is intended to illustrate the visibility of updates in the resultset when using cursors, the exact circumstances under which visibility occurs depends on several factors. These include the size of the resultset relative to the size of the memory buffer, the transaction isolation level, and the frequency with which you commit data, etc.

Case 2 shows how cursor behavior is limited using SQLSetPos when the cursor scrolls backward within a rowset or the cursors move back and forth within a different rowset after an update is applied.

Case 1

Following is an example that shows cursor behavior using positioned operations and shows how positioned updates can be visible to users.

Assume an application performs the following steps:

1. Sets the commit mode to SQL_AUTOCOMMIT_OFF.

   This is a requirement described in "Cursors and autocommit" on page 31.
2. Sets the rowset size to 5.
3. Executes a query to generate a resultset of n rows.
4. Fetches the first rowset of 5 rows with SQLFetchScroll.

A sample resultset is shown below. In the sample, the resultset has only 1 column (defined as varchar(32)). In the table below, the first column shows the row number maintained by the driver internally. The second column shows the actual row values.

Table 10. A sample resultset

Row Counter Stored Internally by the Driver	Row Value
1	Antony
2	Ben
3	Charlie
4	David
5	Edgar

Assume now that the application calls SQLSetPos to update the third and fourth rows of the resultset with the names Caroline and Debbie. After the updates, the actual row values now contain Caroline and Debbie, as shown below:

Table 11. A Sample Resultset

Row Counter Stored Internally by the Driver	Row Value
1	Antony
2	Ben
3	Caroline
4	Debbie
5	Edgar

Note: In some cases, the resultset for a SELECT statement may be too large to fit in memory. As the user scrolls back and forth within the resultset, the ODBC Driver may discard some rows from memory and read in others. This can cause unexpected results: in some situations, updates to data in the cursor may seem to "disappear" and then "reappear" if the cursor re-reads (for example, from disk) the original values for a row that it previously modified.

Case 2

Case 2 shows the limitations when using positioned operations. The following example shows cursor behavior using positioned operations and shows when position updates are not visible to users.

Assume an application performs the following steps:

1. Sets the commit mode to SQL_AUTOCOMMIT_OFF.

This is a requirement explained in “Cursors and autocommit” on page 31.
2. Sets the rowset size to 5.
3. Executes a query to generate a resultset of n rows.
4. Fetches the first rowset of 5 rows with SQLFetchScroll.

   A sample resultset is shown below. In the sample, the first two rowsets are shown. The resultset has only 1 column (defined as varchar(32)). In the table below, the first column shows the row number maintained by the driver internally. The second column shows the actual row values.

Table 12. A Sample Resultset

Row Counter Stored Internally by the Driver	Row Value
1	Antony
2	Ben
3	Charlie
4	David
5	Edgar
6	Fred
7	Gough
8	Harry
9	Ivor
10	John

   Assume that after the first 4 steps above, the application calls SQLSetPos to perform the following tasks:
5. Updates the third row of the resultset.
6. Scrolls to the next rowset by calling SQLFetchScroll. This will get rows 6 to 10 and the cursor will be pointing to row 6.
7. Scrolls backward one rowset to get to the first rowset. This is done by calling SQLScrollFetch with the FETCH_PRIOR option.

After these tasks are performed, the value of the third row that was updated in step 5 still has the old value rather than the updated value as in "Case 1". The updated value is only visible in the Case 2 situation when the change is committed. But due to the unpredictable behavior when setting SQL_AUTOCOMMIT_ON as described in section “Positioned Cursor Operations and SQL_AUTOCOMMIT_ON” on page 32, commits cannot be done until all work related to block cursors and positioned operations is completed.

Using bookmarks

A bookmark is a 32-bit value that an application uses to return to a row. solidDB provides no support for bookmarks.

Error text format

Error messages returned by SQLError come from two sources: data sources and components in an ODBC connection. The error text must use a specific format depending of where the error is issued.

Typically, data sources do not directly support ODBC. Consequently, if a component in an ODBC 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.

If the source of an error is the component itself, the error message must explain this. Therefore, the error text returned by SQLError has two different formats: one for errors that occur in a data source and one for errors that occur in other components in an ODBC connection.

For errors that do not occur in a data source, the error text must use the format:

```
[vendor_identifier][ODBC_component_identifier]
component_supplied_text
```

For errors that occur in a data source, the error text must use the format:

```
[vendor_identifier][ODBC_component_identifier]
[data_source_identifier] data_source_supplied_text
```

The following table shows the meaning of each element.

Table 13. Errors in a Data Source

Element	**Meaning**
vendor_identifier	Identifies the vendor of the component in which the error occurred or that received the error directly from the data source.
ODBC_component_identifier	Identifies the component in which the error occurred or that received the error directly from the data source.
data_source_identifier	Identifies the data source. For single-tier drivers, this is typically a file format. For multiple-tier drivers, this is the DBMS product.
component_supplied_text	Generated by the ODBC component.
data_source_supplied_text	Generated by the data source.

Note: The brackets ([ ]) are included in the error text; they do not indicate optional items.

Sample error messages

The following examples show how various components in an ODBC connection might generate the text of error messages and how solidDB returns them to the application with SQLError.

Table 14. Sample Error Messages

SQLSTATE	Error Message
01000	General warning
01S00	Invalid connection string attribute
08001	Client unable to establish connection

SQLSTATE values are strings that contain five characters; the first two are a class value, followed by a three-character subclass value. For example 01000 has 01 as its class value and 000 as its subclass value. Note that a subclass value of 000 means there is no subclass for that SQLSTATE. Class and subclass values are defined in SQL-92.

Table 15. SQLSTATE values

Class value	Meaning
01	Indicates a warning and includes a return code of SQL_SUCCESS_WITH_INFO.
07, 08, 21, 22, 25, 28, 34, 3C, 3D, 3F, 40, 42, 44, HY	Indicates an error that includes a return value of SQL_ERROR.
IM	Indicates warning and errors that are derived from ODBC.

Processing error messages

Applications provide users with all the error information available through SQLError: the ODBC SQLSTATE, the native error code, the error text, and the source of the error.

The application may parse the error text to separate the text from the information identifying the source of the error. It is the application's responsibility to take appropriate action based on the error or provide the user with a choice of actions.

The ODBC interface provides functions that terminate statements, transactions, and connections, and free statement, connection, and environment handles.

Terminating transactions and connections

The ODBC interface provides functions that terminate statements, transactions, and connections, and free statement (hstmt), connection (hdbc), and environment (henv) handles.

Terminating Statement Processing

To free resources associated with a statement handle, an application calls SQLFreeStmt with the following options:

- SQL_CLOSE - Closes the cursor, if one exists, and discards pending results. The application can use the statement handle again later. In ODBC 3.51, SQLCloseCursor can also be used.

- SQL_UNBIND - Frees all return buffers bound by SQLBindCol for the statement handle.
- SQL_RESET_PARAMS - Frees all parameter buffers requested by SQLBindParameter for the statement handle.

SQLFreeHandle is used to close the cursor if one exists, discard pending results, and free all resources associated with the statement handle.

Terminating transactions

An application calls SQLEndTran to commit or roll back the current transaction.

Terminating connections

To terminate a connection to a driver and data source, an application performs the following steps:

1. Calls SQLDisconnect to close the connection. The application can then use the handle to reconnect to the same data source or to a different data source.
2. Calls SQLFreeHandle to free the connection or environment handle and free all resources associated with the handle.

Constructing an application

This section provides two examples of C-language source code for applications: an example that uses static SQL functions to create a table, add data to it, and select the inserted data; and another example of interactive, ad-hoc query processing.

Microsoft provides two types of header files, one for ASCII data and the other for Unicode data. This example can use either of the Microsoft ODBC header files.

Static SQL example

The following example constructs SQL statements within the application.

```
/*************************************************************
    Sample Name: Example1.c
    Author     : IBM

    Location   : CONSTRUCTING AN APPLICATION-
                 Programmer Guide
    Purpose    : Sample example that uses static SQL
                 functions to
                 create a table,
                 add data to it and
                 select the inserted data.

*************************************************************/
#if (defined(SS_UNIX) || defined(SS_LINUX))
#include <sqlunix.h>
#else
#include <windows.h>
#endif


#if SOLIDODBCAPI
#include <sqlucode.h>
#include <wchar.h>
#else
#include <sql.h>
#include <sqlext.h>
```

```
#endif

#include <stdio.h>
#include <test_assert.h>

#define MAX_NAME_LEN 50
#define MAX_STMT_LEN 100

/***************************************************************
    Function Name: PrintError
    Purpose......: To Display the error associated with
                   the handle
***************************************************************/
SQLINTEGER PrintError(SQLSMALLINT handleType,SQLHANDLE handle)
{
        SQLRETURN  rc = SQL_ERROR;
        SQLWCHAR   sqlState[6];
        SQLWCHAR   eMsg[SQL_MAX_MESSAGE_LENGTH];
        SQLINTEGER nError;

        rc = SQLGetDiagRecW(handleType, handle, 1,
             (SQLWCHAR *)&sqlState, (SQLINTEGER *)&nError,
             (SQLWCHAR *)&eMsg, 255, NULL);
        if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO) {
            printf("\n\t Error:%ls\n",eMsg);
        }
        return(SQL_ERROR);
}


/***************************************************************
        Function Name: DrawLine
        Purpose      : To Draw a specified charcter (chr) for
                       specified number of times (len)
***************************************************************/
void DrawLine(SQLINTEGER len, SQLCHAR chr)
{
        printf("\n");
        while(len > 0)  {
            printf("%c",chr);
            len--;
        }
        printf("\n");

}


/****************************************************************
    Function Name: example1
    Purpose      : Connect to the specified data source and
                   execute the set of SQL Statements
***************************************************************/
SQLINTEGER example1(SQLCHAR *server, SQLCHAR *uid, SQLCHAR *pwd)
{
        SQLHENV     henv;
        SQLHDBC     hdbc;
        SQLHSTMT    hstmt;
        SQLRETURN   rc;

        SQLINTEGER  id;
        SQLWCHAR    drop[MAX_STMT_LEN];
        SQLCHAR     name[MAX_NAME_LEN+1];
        SQLWCHAR    create[MAX_STMT_LEN];
        SQLWCHAR    insert[MAX_STMT_LEN];
        SQLWCHAR    select[MAX_STMT_LEN];
        SQLINTEGER  namelen;
```

```
/* Allocate environment and connection handles. */
/* Connect to the data source. */
/* Allocate a statement handle. */

rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
     &henv);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_ENV,henv));

rc = SQLSetEnvAttr(henv,SQL_ATTR_ODBC_VERSION,
     (SQLPOINTER)SQL_OV_ODBC3,SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_ENV, henv));

rc = SQLAllocHandle(SQL_HANDLE_DBC,henv,&hdbc);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_ENV, henv));

rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS,
     pwd, SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

rc = SQLAllocHandle(SQL_HANDLE_STMT,hdbc,&hstmt);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

    /* drop table 'nameid' if exists, else continue*/
wcscpy(drop, L"DROP TABLE NAMEID");
printf("\n%ls", drop);
DrawLine(wcslen(drop), '-');

rc = SQLExecDirectW(hstmt, drop, SQL_NTS);
if (rc == SQL_ERROR)  {
    PrintError(SQL_HANDLE_STMT, hstmt);
    }

/* commit work*/
rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

/* create the table nameid(id integer,name varchar(50))*/
wcscpy(create,
      L"CREATE TABLE NAMEID(ID INT,NAME VARCHAR(50))");
printf("\n%ls",create);
DrawLine(wcslen(create),'-');

rc = SQLExecDirectW(hstmt,create,SQL_NTS);
if (rc == SQL_ERROR)
    return(PrintError(SQL_HANDLE_STMT,hstmt));

/* commit work*/
rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

/* insert data through parameters*/
wcscpy(insert, L"INSERT INTO NAMEID VALUES(?,?)");
printf("\n%ls", insert);
DrawLine(wcslen(insert), '-');

rc = SQLPrepareW(hstmt, insert, SQL_NTS);
if (rc == SQL_ERROR)
    return(PrintError(SQL_HANDLE_STMT,hstmt));
```

```
/* integer(id) data binding*/
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
         SQL_C_LONG, SQL_INTEGER, 0, 0, &id, 0, NULL);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));


/* char(name) data binding*/
rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT,
        SQL_C_CHAR, SQL_VARCHAR, 0, 0, &name,
        sizeof(name), NULL);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

id = 100;
strcpy(name, "SOLID");

rc = SQLExecute(hstmt);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

/* commit work*/
rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

/* free the statement buffers*/
rc = SQLFreeStmt(hstmt, SQL_RESET_PARAMS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_STMT, hstmt));

rc = SQLFreeStmt(hstmt, SQL_CLOSE);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_STMT, hstmt));

/* select data from the table nameid*/
wcscpy(select, L"SELECT * FROM NAMEID");
printf("\n%ls", select);
DrawLine(wcslen(select), '-');

rc = SQLExecDirectW(hstmt, select, SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

/* bind buffers for output data*/
id = 0;
strcpy(name, "");

rc = SQLBindCol(hstmt, 1, SQL_C_LONG, &id, 0, NULL);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, &name,
     sizeof(name), &namelen);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

rc = SQLFetch(hstmt);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

printf("\n Data ID      :%d", id);
printf("\n Data Name    :%s(%d)\n", name, namelen);

rc = SQLFetch(hstmt);
assert(rc == SQL_NO_DATA);
```

```
        /* free the statement buffers*/
        rc = SQLFreeStmt(hstmt, SQL_UNBIND);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_STMT, hstmt));

        rc = SQLFreeStmt(hstmt, SQL_CLOSE);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_STMT,hstmt));

        /* Free the statement handle. */
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_STMT, hstmt));

        /* Disconnect from the data source. */
        rc = SQLDisconnect(hdbc);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_DBC, hdbc));

        /* Free the connection handle. */
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_DBC, hdbc));

        /* Free the environment handle. */
        rc = SQLFreeHandle(SQL_HANDLE_ENV,henv);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_ENV, henv));

        return(0);
}


/****************************************************************
        Function Name: main
        Purpose      : To Control all operations
****************************************************************/
void main(SQLINTEGER argc, SQLCHAR *argv[])
{
        puts("\n\t SOLID ODBC Driver 3.51:");
        puts("\n\t -Usage of static SQL functions");
        puts("\n\t ==============================");

        if (argc != 4){
            puts("USAGE: Example1 <DSN name> <username> <passwd>");
            exit(0);
            }
        else {
            example1(argv[1], argv[2], argv[3]);
            }
}
```

Interactive ad hoc query example

The following example illustrates how an application can determine the nature of the resultset prior to retrieving results.

```
/****************************************************************
    Sample Name     : Example2.c(ad-hoc query processing)
    Author          : IBM SOLID Information Technology Ltd.

    Location        : CONSTRUCTING AN APPLICATION-
                      Programmer Guide
    Purpose         : To illustrate how an application determines
                      the nature of the result set prior to
                      retrieving results.
```

```
**************************************************************/
#if (defined(SS_UNIX) || defined(SS_LINUX))
#include <sqlunix.h>
#else
#include <windows.h>
#endif

#if SOLIDODBCAPI
#include <sqlucode.h>
#include <wchar.h>
#else
#include <sql.h>
#include <sqlext.h>
#endif

#include <stdio.h>

#ifndef TRUE
#define TRUE 1
#endif

#define MAXCOLS 100
#define MAX_DATA_LEN 255

SQLHENV  henv;
SQLHDBC  hdbc;
SQLHSTMT hstmt;


/**************************************************************
    Function Name: PrintError
    Purpose      : To Display the error associated with
                   the handle
**************************************************************/
SQLINTEGER PrintError(SQLSMALLINT handleType, SQLHANDLE handle)
{
    SQLRETURN rc = SQL_ERROR;
    SQLCHAR sqlState[6];
    SQLCHAR eMsg[SQL_MAX_MESSAGE_LENGTH];
    SQLINTEGER nError;

        rc = SQLGetDiagRec(handleType, handle, 1,
             (SQLCHAR *)&sqlState, (SQLINTEGER *)&nError,
             (SQLCHAR *)&eMsg, 255, NULL);
        if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)  {
            printf("\n\t Error:%s\n",eMsg);
        }
        return(SQL_ERROR);
}


/*************************************************************
    Function Name: DrawLine
    Purpose      : To Draw a specified character (line) for
                   specified number of times (len)
*************************************************************/
void DrawLine(SQLINTEGER len, SQLCHAR line)
{
        printf("\n");
        while(len > 0) {
            printf("%c",line);
            len--;
        }
        printf("\n");

}
```

```
/*************************************************************
    Function Name: example2
    Purpose      : Connect to the specified data source and
                   execute the given SQL statement.
*************************************************************/
SQLINTEGER example2(SQLCHAR *sqlstr)
{
    SQLINTEGER i;

    SQLCHAR colname[32];
    SQLSMALLINT coltype;
    SQLSMALLINT colnamelen;
    SQLSMALLINT nullable;
    SQLINTEGER collen[MAXCOLS];
    SQLSMALLINT scale;
    SQLINTEGER outlen[MAXCOLS];
    SQLCHAR data[MAXCOLS][MAX_DATA_LEN];
    SQLSMALLINT nresultcols;
    SQLINTEGER rowcount, nRowCount=0, lineLength=0;
    SQLRETURN rc;


    printf("\n%s",sqlstr);
    DrawLine(strlen(sqlstr),'=');

    /* Execute the SQL statement. */
    rc = SQLExecDirect(hstmt, sqlstr, SQL_NTS);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_STMT, hstmt));


    /* See what kind of statement it was. If there are */
    /* no result columns, the statement is not a SELECT */
    /* statement. If the number of affected rows is */
    /* greater than 0, the statement was probably an */
    /* UPDATE, INSERT, or DELETE statement, so print */
    /* the number of affected rows. If the number of */
    /* affected rows is 0, the statement is probably a */
    /* DDL statement, so print that the operation was */
    /* successful and commit it. */

    rc = SQLNumResultCols(hstmt, &nresultcols);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_STMT, hstmt));

    if (nresultcols == 0) {
        rc = SQLRowCount(hstmt, &rowcount);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)  {
            return(PrintError(SQL_HANDLE_STMT, hstmt));
            }
        if (rowcount > 0 ) {
            printf("%ld rows affected.\n", rowcount);
            }
        else {
            printf("Operation successful.\n");
            }

        rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_DBC, hdbc));

        }
    /* Otherwise, display the column names of the result */
    /* set and use the display_size() function to */
    /* compute the length needed by each data type. */
```

```
    /* Next, bind the columns and specify all data will */
    /* be converted to char. Finally, fetch and print */
    /* each row, printing truncation messages as */
    /* necessary. */
    else {
        for (i = 0; i < nresultcols; i++) {
            rc = SQLDescribeCol(hstmt, i + 1, colname,
                (SQLSMALLINT)sizeof(colname),
                &colnamelen, &coltype, &collen[i],
                &scale, &nullable);
            if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO){
                return(PrintError(SQL_HANDLE_STMT, hstmt));
                }
            /* print column names */
            printf("%s\t", colname);
            rc = SQLBindCol(hstmt, i + 1, SQL_C_CHAR,
                data[i], sizeof(data[i]), &outlen[i]);
            if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO){
                return(PrintError(SQL_HANDLE_STMT, hstmt));
                }
            lineLength += 6 + strlen(colname);
            }

        DrawLine(lineLength-6,'-');

        while (TRUE) {
            rc = SQLFetch(hstmt);
            if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO){
                nRowCount++;
                for (i = 0; i < nresultcols; i++) {
                    if (outlen[i] == SQL_NULL_DATA) {
                        strcpy((char *)data[i], "NULL");
                        }
                    printf("%s\t",data[i]);
                    }
                printf("\n");
                }

            else {
                if (rc == SQL_ERROR)
                    PrintError(SQL_HANDLE_STMT, hstmt);
                break;
                }
            }
        printf("\n\tTotal Rows:%d\n", nRowCount);
        }

    SQLFreeStmt(hstmt, SQL_UNBIND);
    SQLFreeStmt(hstmt, SQL_CLOSE);
    return(0);

}


/****************************************************************
    Function Name: main
    Purpose      : To Control all operations
***************************************************************/
int __cdecl main(SQLINTEGER argc, SQLCHAR *argv[])
{
    SQLRETURN   rc;

    printf("\n\t SOLID ODBC Driver 3.51-Interactive");
    printf("\n\t ad-hoc Query Processing");
    printf("\n\t =================================\n");

    if (argc != 4)  {
```

```
        puts("USAGE: Example2 <DSN name> <username> <passwd>");
        exit(0);
        }


    /* Allocate environment and connection handles. */
    /* Connect to the data source. */
    /* Allocate a statement handle. */
    rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_ENV, henv));

    rc = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER)SQL_OV_ODBC3, SQL_NTS);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_ENV, henv));

    rc = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_ENV, henv));

    printf("\n Connecting to %s\n ", argv[1]);
    rc = SQLConnect(hdbc, argv[1], SQL_NTS, argv[2], SQL_NTS,
        argv[3], SQL_NTS);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

    rc = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

    /* execute the following SQL statements */
    example2("SELECT * FROM SYS_TABLES");
    example2("DROP TABLE TEST_TAB");
    example2("CREATE TABLE TEST_TAB(F1 INT, F2 VARCHAR)");
    example2("INSERT INTO TEST_TAB VALUES(10, 'SOLID')");
    example2("INSERT INTO TEST_TAB VALUES(20, 'MVP')");
    example2("UPDATE TEST_TAB SET F2='UPDATED' WHERE F1 = 20");
    example2("SELECT * FROM TEST_TAB");


    /* Free the statement handle. */
    rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_STMT, hstmt));

    /* Disconnect from the data source. */
    rc = SQLDisconnect(hdbc);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

    /* Free the connection handle. */
    rc = SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

    /* Free the environment handle. */
    rc = SQLFreeHandle(SQL_HANDLE_ENV, henv);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_ENV, henv));

    return(0);

}
```

Testing and debugging an application

The Microsoft ODBC SDK provides tools for application development.

The tools included are as follows:

- ODBC Test, an interactive utility that enables you to perform ad hoc and automated testing on drivers. A sample test DLL (the Quick Test) is included, which covers basic areas of ODBC driver conformance.
- ODBC Spy, a debugging tool with which you can capture data source information, emulate drivers, and emulate applications.
- Sample applications, including source code and makefiles:
  - A #define, ODBCVER, to specify which version of ODBC you want to compile your application with. To use the ODBC 3.51 constants and prototypes, add the following line to your application code before providing the include files.

    ```
    #define ODBCVER 0X0352
    ```
  - For ASCII data, use the following standard Microsoft include files:

    `SQL.H` and `SQLEXT.H`
  - For Unicode data, use the following Microsoft include files:

    `SQLUCODE.H` and `WCHAR.H`

For additional information about the ODBC SDK tools, see the *Microsoft ODBC SDK Guide*.

Installing and configuring ODBC software

When the solidDB Development Kit is installed, it automatically installs the ODBC drivers and a number of user Data Source Names (DSN). You can also add you own user DSNs.

For details about configuring and connecting to an ODBC data source, read "Connecting to a data source" on page 13.

Application developers must decide whether to redistribute these programs or write their own setup and administration programs. For more information about the Driver Setup Toolkit and the ODBC Administrator, see the *Microsoft ODBC SDK Guide* on the Microsoft Web site.

A setup program written by an application developer uses the installer DLL to retrieve information from the ODBC.INF file, which is created by a driver developer and describes the disks on which the ODBC software is shipped. The setup program also uses the installer DLL to retrieve the target directories for the Driver Manager and the drivers, record information about the installed drivers, and install ODBC software.

Administration programs written by application developers use the installer DLL to retrieve information about the available drivers, to specify default drivers, and to configure data sources.

Application developers who write their own setup and administration programs must ship the installer DLL and the `ODBC.INF` file.

With the current version of ODBC 3.51, the Installer for Windows does not contain the Microsoft Driver Manager. To maintain compatibility with ADO, OLE DB, and

ODBC, Microsoft recommends obtaining the Driver Manager and installing it. To do this, users need to download the executable mdac_typ.exe from the Microsoft Web site and install it; this executable provides users with Driver Manager 3.5 or above. For the URL to the Microsoft Web site where this executable is found, refer to the IBM Corporation Web site or the Release Notes.

3 Using Unicode

This section describes how to implement the Unicode standard, providing the capability to encode characters used in the major languages of the world.

What is Unicode?

The Unicode Standard is the universal character encoding standard used for representation of text for computer processing. Unicode provides a consistent way of encoding multilingual plain text making it easier to exchange text files internationally.

The version 2.0 Unicode Standard is fully compatible with the International Standard ISO/IEC 10646-1; 1993, and contains all the same characters and encoding points as ISO/IEC 10646. This code-for-code identity is true for all encoded characters in the two standards, including the East Asian (Han) ideographic characters. The Unicode Standard also provides additional information about the characters and their use. Any implementation that conforms to Unicode also conforms to ISO/IEC 10646.

Unicode uses a 16-bit encoding that provides code points for more than 65,000 characters. To keep character coding simple and efficient, the Unicode Standard assigns each character a unique 16-bit value, and does not use complex modes or escape codes.

While 65,000 characters are sufficient for encoding most of the many thousands of characters used in major languages of the world, the Unicode standard and ISO 10646 provide an extension mechanism called UTF-16 that allows for encoding as many as a million more characters, without use of escape codes. This is sufficient for all known character encoding requirements, including full coverage of all historic scripts of the world.

What characters does the Unicode standard include

The Unicode Standard defines codes for characters used in the major languages written today. This includes punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, dingbats, etc. In all, the Unicode Standard provides codes for nearly 39,000 characters from the world's alphabets, ideograph sets, and symbol collections.

There are about 18,000 unused code values for future expansion in the basic 16-bit encoding, plus provision for another 917,504 code values through the UTF-16 extension mechanism. The Unicode Standard also reserves 6,400 code values for private use, which software and hardware developers can assign internally for their own characters and symbols. UTF-16 makes another 131,072 private use code values available, should 6,400 be insufficient for particular applications.

Encoding forms

Character encoding standards define not only the identity of each character and its numeric value, or code position, but also how this value is represented in bits. The Unicode Standard endorses two forms that correspond to ISO 10646 transformation formats, UTF-8 and UTF-16.

The ISO/IEC 10646 transformation formats UTF-8 and UTF-16 are essentially ways of turning the encoding into the actual bits that are used in implementation.

- UTF-16

  UTF-16 assumes 16-bit characters and allows for a certain range of characters to be used as an extension mechanism in order to access an additional million characters using 16-bit character pairs. The Unicode Standard, Version 2.0, has adopted this transformation format as defined in ISO/IEC 10646.
- UTF-8

  UTF-8 is a way of transforming all Unicode characters into a variable length encoding of bytes. It has the advantages that the Unicode characters corresponding to the familiar ASCII set end up having the same byte values as ASCII, and that Unicode characters transformed into UTF-8 can be used with much existing software without extensive software rewrites. The Unicode Consortium also endorses the use of UTF-8 as a way of implementing the Unicode Standard. Any Unicode character expressed in the 16-bit UTF-16 form can be converted to the UTF-8 form and back without loss of information.

For storing Unicode data internally,solidDB uses a subset of UTF-16 known as UCS-2 (two-octet encoding). With this encoding, all the characters in the so called Basic Multilingual Plane of UTF-16 are properly stored and processed. Almost all modern languages are encoded in BMP (including all modern Asian character sets). Characters that are outside of the BMP (known also as surrogate code points or supplementary characters), are properly stored but some of the string functions and length properties may produce incorrect results. For all practical purposes, solidDB supports externally UTF-16 and UTF-8 encoding of Unicode.

Implementing Unicode

This section contains information required to implement the Unicode standard in solidDB.

Note the following implementation guidelines:

- Unicode data types

  SQL data types WCHAR, WVARCHAR and LONG WVARCHAR are used to store Unicode data in a solidDB database. The Wide-character implementation conforms to ODBC 3.5 specification. The Unicode data types are also interoperable with corresponding character data types (CHAR, VARCHAR and LONG VARCHAR), but conversions from Unicode data types to character data types fail, if the characters are beyond ISO Latin 1. All string operations are possible between Unicode and character data types with implicit type conversions.
- Use of UTF-16 encoding

  If the data is externally encoded in UTF-16, no conversion is needed while transferring the data to and from the server. In ODBC, C type wchar_t* type should be used to store the data, and the C type identifier SQL_C_WCHAR should be used to bind the variables to the column types WCHAR and derivatives. In JDBC, the processing is automatic, independently of the code page set.
- Use of UTF-8 encoding

  If the data is externally encoded in UTF-8, it has to be converted to the internal representation. In JDBC, the conversion is automatic once the code page is set to

UTF-8. In ODBC, the conversion is enacted for data stored in variables of type char* while binding to the WCHAR (or derivatives) columns. The are two alternative ways of doing that:

1. Make changes in the server or client-side parameter values to set the conversions to/from UNICODE data types (WCHAR/WVARCHAR/LONG WVARCHAR) to char (denoted SQL_C_CHAR in ODBC) to expect UTF-8 encoding in the application data buffer.
   - In the server-side solid.ini, set the **Srv.ODBCDefaultCharBinding** parameter to UTF-8 (default is Raw).

     This will enable the UTF-8 mode for all the connections to the server.

     ```
     [Srv]
     ODBCDefaultCharBinding=UTF-8
     ```
   - In the client-side solid.ini, set the **Client.ODBCCharBinding** parameter to UTF-8 (default is Raw).

     The UTF-8 mode will be enabled for the applications using the client-side solid.ini in question.

     ```
     [Client]
     ODBCCharBinding=UTF-8
     ```
2. In the application, force the binding to UTF-8 by using solidDB specific SQL_C_UTF8, instead of the standard SQL_C_CHAR C data type. Note that the SQL_C_UTF8 is only visible to application if it includes *solidodbc3.h* instead of standard ODBC header files.

- Internal storage format

  In solidDB, the storage format for Unicode column data is UCS-2. All character information in the data dictionary are stored as Unicode.

  The wide character types require more storage space than normal character types. Therefore, use wide characters only where necessary.
- Ordering data columns

  Unicode data columns are ordered based on the binary values of the UCS-2 format. If the binary order is different than what natural language users expect, developers need to provide a separate column to store the correct ordering information.
- Unicode file names

  A solidDB server does not support using Unicode strings in any file names.

Setting up Unicode data

This section contains information on how to setup Unicode data in solidDB.

Creating columns for storing Unicode data

In order to start storing Unicode data in a solidDB database, tables with Unicode data columns need to be created first as follows:

```
CREATE TABLE customer (c_id INTEGER, c_name WVARCHAR,...)
```

Loading Unicode data

You can use the data import tool Speed Loader to import data to Unicode columns. The import files should contain Unicode data in UTF-8 format.

Using Unicode in database entity names

It is possible to name tables, columns, procedures, etc. with Unicode strings, simply by enclosing the Unicode names with double quotes in all the SQL statements.

solidDB tools will handle Unicode strings in UTF-8 format. In order to enter native Unicode strings, third-party database administration applications need to be used, or a special application using solidDB JDBC Driver should be written for this purpose.

Unicode user names and passwords

User names and passwords can also be Unicode strings. However, to avoid access problems from different tools, the original database administrator account information must be given as pure ASCII strings.

solidDB Data Dictionary, Export, and solidDB Speed Loader

The solidDB tools use UTF-8 as the external representation format of Unicode strings.

solidDB Speed Loader (solload) accepts Unicode data in control and input files in UTF-8 format.

Export (solexp) extracts Unicode data from database to output files in UTF-8 format.

solidDB Data Dictionary (soldd) prints table, column, etc. names containing Unicode strings in UTF-8 format into the SQL DDL file.

Note that the teletype SQL Editor (solsql) can use the SQL files output by soldd to create the tables, indices, etc. for a new database, as well as data definition entries if Unicode strings are available for them.

Data Dictionary and Export accept option -8 to allow exporting data dictionary information in 8-bit format. The option -8 is needed if there are Scandinavian or other national non-ascii characters in the data dictionary names.

Teletype tools

The teletype versions of solidDB SQL Editor (solsql) and Remote Control (solcon) will function correctly in Unicode client environments.

Unicode and ODBC Driver

The solidDB ODBC Driver, which conforms to the Microsoft ODBC 3.51 standard, is Unicode compliant.

Note: solidDB has two ODBC drivers available, one with Unicode support, and one for ASCII-only use.

Old client versions

Old clients can connect to solidDB. All Unicode data is converted to ISO Latin 1 whenever possible. Thus, provided only ISO-Latin 1 data is used in the database, old clients can access solidDB.

Note: To avoid future problems, it is recommended that you upgrade your client applications to use current client libraries.

Unicode variables and binding

Using string columns containing Unicode data work just like normal character columns. Note that the length of string buffers is given as the number of bytes required to store the value.

String functions

String functions work as expected, also between ISO Latin 1 and Unicode strings. Conversions are provided implicitly, when necessary. The result is always of Unicode type, if either of the operands is Unicode.

The functions UPPER() and LOWER() work on Unicode strings when the contained characters can be mapped to ISO Latin 1 code page.

Translations

The character translations defined in client side `solid.ini` do not affect the data stored in Unicode columns. Translations remain in effect for character columns.

Unicode and solidDB JDBC Driver

Unicode is supported in the solidDB JDBC Driver, an solidDB implementation of the JDBC 2.0 standard. It is also compatible with solidDB, solidDB Database Engine 4.x.

As Java uses natively Unicode strings, supporting Unicode means primarily that when accessing Unicode columns in solidDB, no data type conversions are necessary. Additionally, JDBC ResultSet Class methods getUnicodeStream and setUnicodeStream are supported now for handling large Unicode texts stored in solidDB.

To convert Java applications to support Unicode, the string columns in solidDB need to be redefined with Unicode data types.

4 Using the solidDB JDBC Driver

This section describes how to use the solidDB JDBC Driver.

The solidDB JDBC Driver 2.0 is a JDBC type 4 driver. *Type 4* means that this is a 100% Pure Java implementation of the Java Database Connectivity (JDBC) 2.0 standard.

What is solidDB JDBC Driver

This topic provides information about the solidDB JDBC driver.

The JDBC API defines Java classes to represent database connections, SQL statements, result sets, database metadata, etc. It allows a Java programmer to issue SQL statements and process the results. JDBC is the primary API for database access in Java. More information on the JDBC technology can be found at the JDBC Technology Homepage (http://java.sun.com/products/jdbc/).

solidDB's JDBC driver is written entirely in Java and communicates directly with the solidDB server using the TCP/IP network protocol. The solidDB driver does not require any additional database access libraries, such as ODBC. The driver requires that a JRE (Java Runtime Environment) or JDK (Java Development Kit) is available.

The solidDB JDBC Driver is a solidDB implementation of the JDBC 2.0 standard. It is usable in all Java environments supporting JDK 1.4.2 and above.

Getting started with solidDB JDBC Driver

This topic describes how to get started with solidDB JDBC Driver.

To get started with solidDB JDBC Driver, be sure you have:

1. Installed the JDBC Driver and verified the installation. For details, follow the instructions in the JDBC Readme in the solidDB installation directory.
2. Set up the development environment so that it supports JDBC properly. solidDB JDBC Driver expects support for JDBC version 2.0x. The JDBC interface is included in the `java.sql` package. To import this package, be sure to include the following line in the application program:

   ```
   import java.sql.*;
   ```

Registering solidDB JDBC Driver

The JDBC driver manager handles loading and unloading drivers and interfacing connection requests with the appropriate driver.

The driver can be registered as shown below. After execution of this code, the driver registers itself in the DriverManager.

```
// registration using Class.forName service
Class.forName("solid.jdbc.SolidDriver");
```

See the source code for the Sample 1 application in “Code examples” on page 78.

Connecting to the database

Once the driver is successfully registered with the driver manager, a connection is established by creating an instance of java.sql.Connection with the following code.

The parameter required by the DriverManager.getConnection function is the JDBC connection string. The JDBC connection string identifies which computer the database server is running on; the string also contains other information required to connect to the server.

```
Connection conn = null;
// sCon is the JDBC connection string
(jdbc:solid://hostname:port/login/password)
String sCon = "jdbc:solid://fb9:1314/dba/dba";
try {
    conn =  DriverManager.getConnection(sCon);
} catch (SQLException e) {
    System.out.println("Connect failed : " + e.getMessage());
}
```

The syntax of the JDBC URL (connection string) for solidDB is:

```
jdbc:solid://<hostname>:<port>/<username>/<password>[?<property-name>=<value>]...
```

The string

```
"jdbc:solid://fb9:1314/dba/dba"
```

attempts to connect to a solidDB server in machine fb9 listening to the tcp/ip protocol at port 1314.

The application can establish multiple connections to the database by using multiple Connection objects. Developers should manage connection lifecycle in a very accurate way, otherwise there can be a lot of conflicts between concurrent users and applications trying to access the database. For details and instructions, see “Code examples” on page 78.

In this topic, we describe issues related to the solidDB products. Generic information about the JDBC usage may be found at the Sun Java Developers' website: http://java.sun.com/developer/onlineTraining/distributed/index.html

See the source code for the Sample 1 application in “Code examples” on page 78.

Note: ThesolidDB JDBC Driver only supports a connection for administration options, with no queries allowed. For this type of connection, set the java.util.Properties name ADMIN_ USER to true. After it is set to true and a connection is established, then only ADMIN commands are allowed.

Transactions and autocommit mode

As the JDBC specification defines, a connection to the solidDB database can be in either autocommit or non-autocommit mode. When not in autocommit mode, each transaction needs to be explicitly committed before the modifications it made can be seen by other database connections. The autocommit state can be monitored by Connection.getAutoCommit() method. The state can be set by Connection.setAutoCommit(). An solidDB server's default setting for autocommit state is true. If autocommit mode is off, then the transactions can be committed in two ways.

- calling the Connection.commit() method, or
- executing a statement for SQL 'COMMIT WORK'

Handling database errors

Database errors in JDBC are handled and managed by the exception mechanism, which is one of the Java language's strengths. Most of the methods, specified in JDBC interfaces, may throw an instance of SQLException. As these errors may appear in the normal application workflow (representing concurrency conflicts, for instance) your code should be tolerant to such errors. Basically, you must not leave your connections in any other state than "closed" regardless of the result of your code's execution. This approach allows avoiding situations where all available connections remain open due to unhandled exceptions.

You can get an exception's error code by calling e.getErrorCode(). For native error codes, see the appendix, "Error Codes" in *IBM solidDB Administrator Guide*.

The following code example shows a correct way of handling errors coming from the database:

```
Public void listTablesExample() {
    try {
        Class.forName("solid.jdbc.SolidDriver");
    } catch (ClassNotFoundException e) {
        System.err.println("Solid JDBC driver is not registered
    in the classpath");
        return; //exit from the method
    }
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        conn = DriverManager.getConnection("jdbc:solid://
        localhost:1313", "dba", "dba");
        stmt = conn.createStatement();
        rs = stmt.executeQuery("SELECT * FROM tables");
        while (rs.next()) {
            System.out.println(rs.getObject(0));//printing
        out results
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        /* It's a good idea to release
    resources in a finally{} block
    in reverse-order of their creation
    if they are no-longer needed
    */
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException sqlEx) { // ignore
                rs = null;
            }
        }
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException sqlEx) { // ignore
                stmt = null;
            }
        }
    }
    if (conn != null)
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
```

```
        } finally {
            conn = null;
        }
}
```

Special notes about solidDB and JDBC

JDBC does not really specify what SQL dialect you can use; it simply passes the SQL on to the driver and lets the driver either pass it on directly to the database, or parse the SQL itself. Because of this, the solidDB JDBC Driver behavior is particular to solidDB

In some functions, the JDBC specification leaves some details open. For the details particular to solidDB's implementation of the methods, check “JDBC driver interfaces and methods”.

The solidDB JDBC Driver provides support for catalogs and schemas in solidDB.

Executing Stored Procedures

In solidDB databases, stored procedures can be called by executing statements 'CALL *proc_name* [(*parameter* ...)]' as in any other SQL statement. Procedures can also be used in JDBC in the same way, through a standard CallableStatement interface.

Note: solidDB stored procedures can return result sets. Calling procedures through the JDBC CallableStatement interface is not necessary. For an example of calling solidDB procedures using JDBC, see the source code for the Sample 3 application in “Code examples” on page 78.

JDBC driver interfaces and methods

solidDB JDBC Driver 2.0 is compatible with the JDBC 2.0 standard, with support to selected features of JDBC 2.0 Optional Package (known before as Standard Extension).

This topic describes solidDB-specific differences from the standard API. You can browse standard packages and interfaces in the `java.sql` and `javax.sql` packages, and see details of a particular implementation by checking the list of "All Known Implementing Classes".

For a description of how different data types are supported by solidDB JDBC Driver, see “solidDB JDBC Driver type conversion matrix” on page 90.

Array

The java.sql.Array interface is not supported. This interface is used to map SQL type Array in the Java programming language. It reflects the SQL-99 standard that is currently unavailable in solidDB.

Blob

The java.sql.Blob interface is not supported. This interface is used to map SQL type Blob in the Java programming language. It reflects the SQL-99 standard that is currently unavailable in solidDB.

CallableStatement

A java.sql.CallableStatement interface is intended to support calling database stored procedures. Thus, solidDB stored procedures are used in JDBC in the same way as any statement; the use of class CallableStatement is not necessary when you are writing applications on a solidDB server only. However, for portability reasons, using CallableStatement is a wise choice.

Note: The JDBC Driver allows for the creation of a Statement object that generates ResultSet objects with the given type and concurrency. This differs from the createStatement method in JDBC 1.0 because it allows the default result set type and result set concurrency type to be overridden.

Differences with the standard API

Following are the differences from the standard CallableStatement interface defined in the JDBC API.

Table 16. Differences to the Standard CallableStatement Interface

Method name	Notes
getArray(int i)	Not supported by solidDB.
getBlob(int i)	Not supported by solidDB.
getClob(int i)	Not supported by solidDB.
getDate(int parameterIndex, Calendar cal)	Works as specified in Java API. **Note:** Uses a given Calendar object to specify time zone and locale, different from default ones. The same rule corresponds to other similar methods operating with Calendar instances.
getObject (int i, Map map)	Not supported by solidDB.
getRef(int i)	Not supported by solidDB.
registerOutParameter(int parameterIndex, int sqlType, String typeName)	Not supported by solidDB. This method throws an exception with the following message: "This method is not supported"

Clob

The java.sql.Clob interface is not supported. This interface is used to map SQL type Clob in the Java programming language. It reflects the SQL-99 standard that is currently unavailable in solidDB.

Connection

The java.sql.Connection interface is a public interface. It is used to establish a connection (session) with a specified database. SQL statements are executed and results are returned within the context of a connection.

Differences with the standard API

Following are the differences from the standard Connection interface defined in the JDBC API.

Table 17. Differences to the Standard Connection Interface

Method name	Notes
getTypeMap()	solidDB provides this method, but it always returns null.
isReadOnly()	solidDB only supports read-only connections and read-only transactions if the database is declared as read-only. This method always returns false.
nativeSQL(String sql)	Works as specified in Java API. solidDB JDBC Driver does not change the SQL passed to the solidDB server. The SQL query the user passes is returned.
prepareCall(String sql)	Works as specified in Java API. **Note:** Note that the escape call syntax is not supported.
setReadOnly(boolean readOnly)	solidDB only supports read-only database and read-only transactions if the database is declared as read-only. This method exists but does not affect the connection behavior.
setTransactionIsolation(int level)	Works as specified in Java API.
setTypeMap(Map map)	Not supported by solidDB.

DatabaseMetaData

The java.sql.DatabaseMetaData interface is a public abstract interface. It provides general, comprehensive information about the database.

All methods for this interface are supported by solidDB.

For a description of how different data types are supported by solidDB JDBC Driver, see “solidDB JDBC Driver type conversion matrix” on page 90.

Driver

The java.sql.Driver interface is a public abstract interface. Every driver class implements this interface and all methods for this interface are supported by solidDB.

PreparedStatement

The java.sql.PreparedStatement interface is a public abstract interface. It extends the statement interface. It provides an object that represents a precompiled SQL statement.

Note: The JDBC Driver allows for the creation of a PreparedStatement object that generates ResultSet objects with the given type and concurrency. This differs from the prepareStatement method in JDBC 1.0 because it allows the default result set type and result set concurrency type to be overridden.

Subinterfaces: CallableStatement

Differences with the standard API

Following are the differences from the standard PreparedStatement interface defined in the JDBC API.

Table 18. Differences to the Standard PreparedStatement Interface

Method name	Notes
setArray(int i, Array x)	Not supported by solidDB.
setBlob(int I, Blob x)	Not supported by solidDB.
setClob(int I, Clob x)	Not supported by solidDB.
setObject(int parameterIndex, Object x)	Works as specified in Java API. **Note:** The following objects are not supported by solidDB: BLOB, CLOB, ARRAY, REF, and object (USING java.util.Map).
setObject(int parameterIndex, Object x, int targetSqlType))	Not supported by solidDB. This method throws an exception with the following message: "This method is not supported"
setObject(int parameterIndex, Object x, int targetSQLType, int scale)	Not supported by solidDB. This method throws an exception with the following message: "This method is not supported"
setRef(int I, Ref x)	Not supported by solidDB.

Ref

The java.sql.Ref interface is a public abstract interface.

This interface is a reference to an SQL structured type value in the database. This interface is not supported by solidDB.

ResultSet

The java.sql.ResultSet interface is a table of data that represents a database result set from a query statement. This object includes a cursor that points to its current row of data. The cursor's initial position is before the first row. It is moved to the next row by the next method. When there are no more rows left in the result set, the method returns false; this allows the use of a WHILE loop to iterate through the result set.

Differences with the standard API

Following are the differences from the standard ResultSet interface defined in the JDBC API.

Table 19. Differences to the Standard ResultSet Interface

Method name	Notes
getArray(int i)	Not supported by solidDB.
getArray(String ColName)	Not supported by solidDB.
getBigDecimal(String columnName)	Works as specified in Java API.
getCharacterStream(int columnIndex)	Works as specified in Java API. NOTE: The JDBC Driver sets the designated parameter to the given Reader object at the given character length. When a large UNICODE value is input to a LONG VARCHAR/LONG WVARCHR parameter, for convenience, you can send it via a java.io.Reader. The JDBC Driver reads the data from the stream as needed, until it reaches end-of-file. The driver does all the necessary conversion from UNICODE to the database CHAR format.
getCharacterStream(String columnName)	Works as specified in Java API. The note above also applies to this method.
getFetchSize()	Not supported by solidDB.
getObject(int columnIndex)	Works as specified in Java API. NOTE: following objects are not supported by solidDB: BLOB, CLOB, ARRAY, REF, and object (USING java.util.Map).
getObject(int i, Map map)	Not supported by solidDB.
getObject(String colName, Map map)	Not supported by solidDB. This method throws an exception with the following message: "This method is not supported"
getRef(int i)	Not supported by solidDB.
getRef(String colName)	Not supported by solidDB.
refreshRow()	Not supported by solidDB.
setFetchSize(int rows)	No operation in solidDB. Sets the value for the number of rows to be fetched from the database each time. The value a user sets with this method is ignored.

ResultSetMetaData

The java.sql.ResultSetMetaData interface is a public abstract interface. This interface is used to find out about the types and properties of the columns in a ResultSet.

SQLData

The java.sql.SQLData interface is not supported. This interface is used to custom map SQL user-defined types. It reflects the SQL-99 standard that is currently unavailable in solidDB.

SQLInput

The java.sql.SQLInput interface is not supported. This interface is an input stream that represents an instance of an SQL structured or distinct type. It reflects the SQL-99 standard that is currently unavailable in solidDB.

SQLOutput

The java.sql.SQLOutput interface is not supported. This interface is an output stream used to write the attributes of a user-defined type back to the database. It reflects the SQL-99 standard that is currently unavailable in solidDB.

Statement

The java.sql.Statement interface is a public abstract interface. It is the object used to execute a static SQL statement and obtain the results of the execution.

Note: The JDBC Driver allows for the creation of a Statement object that generates ResultSet objects with the given type and concurrency. This differs from the CreateStatement method in JDBC 1.0 because it allows the default result set type and result set concurrency type to be overridden.

Subinterfaces:
- CallableStatement
- PreparedStatement

Differences with the standard API

Following are the differences from the standard Statement interface defined in the JDBC API.

Table 20. Differences to the Standard Statement Interface

Method name	Notes
getFetchSize()	No operation in solidDB.
getMaxFieldSize()	Maxfield size does not affect the solidDB server's behavior.
getMoreResults()	solidDB does not support multiple result sets.

Table 20. Differences to the Standard Statement Interface (continued)

Method name	Notes
getResultSetType()	Not supported by solidDB.
setFetchSize(int rows)	No operation in solidDB. Sets the value for the number of rows to be fetched from the database each time. The value a user sets with this method is ignored.
setMaxFieldSize(int max)	Maxfield size does not affect the solidDB server's behavior.

Struct

The java.sql.Struct interface is not supported. This interface represents the standard mapping in the Java programming language for an SQL structured type. It reflects the SQL-99 standard that is currently unavailable in solidDB.

ResultSet (updateable)

The java.sql.Resultset interface contains methods for producing ResultSet objects that are updateable. A result set is updateable if its concurrency type is CONCUR_UPDATABLE. Rows in the result set may be updated, deleted, or new rows inserted using methods update xxx, where xxx refers to the datatype and methods updateRow and deleteRow.

Differences with the standard API

Following are the differences from the standard ResultSet interface defined in the JDBC API.

Table 21. Differences to the Standard ResultSet Interface

Method name	Notes
getRef(int i)	This method is not supported.
getRef(String colName)	This method is not supported.
refreshRow()	This method is not supported.
rowDeleted()	This method is not supported.
rowInserted()	This method is not supported.
setFetchSize(int rows)	This method is not supported.

JDBC driver enhancements

This section describes enhancements to the JDBC Driver.

WebSphere compatibility

This section describes JDBC driver features that improve WebSphere® compatibility.

solidDB Data Store Helper Class in WebSphere

WebSphere needs an adapter class for those JDBC data sources that are to be used within WebSphere. The base class for these adapters is the com.ibm.websphere.rsadapter.GenericDataStoreHelper class and solidDB implements its own version of this adapter inside a class called:com.ibm.websphere.rsadapter.SolidDataStoreHelper.

This class, in turn, is provided within the solidDB product as a separate archive file called `SolidDataStoreHelper.jar` You can find this file in the same directory with the JDBC driver jar file in the solidDB product distribution directory tree.

When you are configuring a new solidDB data source in WebSphere, you need to give the classcom.ibm.websphere.rsadapter.SolidDataStoreHelper in the data store helper field of the configuration. Also, you need to specify the full path to the `SolidDataStoreHelper.jar` file in the data source configuration of WebSphere. See the WebSphere documentation for further details how to define new data sources in WebSphere. See also the solidDB 'websphere' sample section for how to install the solidDB's WebSphere sample application in the Websphere Studio Application Developer's workspace.

solidDB Data Source Properties and WebSphere

You need to define the following properties when configuring a new data source in the WebSphere:

URL

- type: java.lang.String
- value should use syntax similar to the following syntax: 'jdbc:solid://<hostname>:<port>'

user

- type: java.lang.String
- value should be a valid user name

password

- type: java.lang.String
- value should be a valid password

Connection timeout in JDBC

This topic describes features available in solidDB to set a connection timeout.

Connection timeout means response timeout of any JDBC call invoking data transmission over a connection socket. If the response message is not received within the time specified, an I/O exception is thrown. The JDBC standard (2.0/3.0) does not support setting of the connection timeout. solidDB has introduced two ways for doing that: one using a non-standard driver manager extension method and the other one using the property mechanisms. The time unit in either case is one millisecond.

Driver Manager Method get/setConnectionTimeout()

The following example illustrates the solution. The effect of the setting is immediate. This allows to set the timeout to zero if you want to force-disconnect.

```
//Import Solid JDBC:
import solid.jdbc.*;

//Define the connection:
solid.jdbc.SolidConnection conn = null;

//Cast to SolidConnection in order to use Solid-specific methods:
conn = (SolidConnection)java.sql.DriverManager.getConnection(sCon);

//Set connection timeout in milliseconds:
conn.setConnectionTimeout(3000);
```

Non-standard connection properties

The following connection properties can be used to attain connection-specific behavior.

Statement cache property

solidDB JDBC driver introduces a property for setting the value of the connection's statement cache.

The name of the property is "StatementCache" and the default size of the cache is 8. The valid value range is 1 to 512 (inclusive). If the value exceeds the range, the driver silently forces the value either to 1 or 512.

Below is an example on how to use the property.

```
// create a Solid JDBC driver instance
Class.forName("solid.jdbc.SolidDriver");

// create a new Properties instance and insert a value for
// StatementCache property
java.util.Properties props = new java.util.Properties();
props.put("StatementCache","32");

// define the connection string to be used
String sCon="jdbc:solid://localhost:1315/uname1/pwd1";

// get the Connection object with a statement cache of 32
java.sql.Connection conn = java.sql.DriverManager.getConnection(sCon, props);
```

Timeout properties

The timeouts listed in the sections below may be set as connection properties.

Connection timeout property

Using the property "solid_connection_timeout_ms", you can set the connection timeout value (in milliseconds). The property must be set before getting a new connection. Once a connection object is created, changing the property value has no effect.

Login timeout property

Using the property "solid_login_timeout_ms", you can set the timeout (in milliseconds) for opening of a connection .

Note: You can use also the method DriverManger.setLoginTimeout(seconds) to set the login timeout. This is a standard-compliant method.

Idle Timeout Property

Using the property "solid_idle_timeout_min", you can set the timeout (in minutes) that fires when a connection has been idle for a longer time than specified (in minutes).

If the property value is not set, the server-side setting of the configuration parameter **ConnectTimeOut** applies. The factory value is 480 minutes. 0 means 'infinite timeout' (never expires).

Example

The following example shows how to set a connect timeout using the "solid_connection_timeout_ms" property.

```
// Set connection timeout with "solid_connection_timeout_ms" property //
public class Test {

  public static void main( String args[] ){

    // create property object
    Properties props = new Properties();

    // put username and password in the properties
    props.put("user", "MYUSERNAME");
    props.put("password", "MYPASSWORD");

    //
    // Put connection timeout in the property object
    //
    props.put("solid_connection_timeout_ms", "10000");

    try {

      // create driver
      Driver d = (Driver)(
          Class.forName("solid.jdbc.SolidDriver").newInstance());

      // get connection with url and property info
      Connection c = DriverManager.getConnection(
          "jdbc:solid://localhost:1313", props );

      // close connection
      c.close();

    } catch ( Exception e ) {
      ; // save the day
    }
  }
}
```

Appinfo property

The connection attribute called Appinfo can be used to uniquely identify applications running in the same computer and under the same username, for the purposes of tracing and management. It can be retrieved, on the server side, with the command `ADMIN COMMAND 'userlist'`. By default, the value (a string) is not set. It is possible to set the value with the connection property "solid_appinfo".

Note: In ODBC applications, the value of Appinfo is passed by way of the environmental variable SOLAPPINFO.

Transparent connectivity (TC) properties

Transparent connectivity (TC) is a connection mode that can be used with solidDB HA solution (HotStandby). In JDBC, the TC mode is enacted with the following connection properties:

- "solid_tf_level"

  Sets the transparent failover level to "CONNECTION or "SESSION", or "NONE. The default is "NONE".
- "solid_preferred_access"

  Sets the preferred access mode to "WRITE_MOSTLY" or "READ_MOSTLY". The value "READ_MOSTLY" enacts automatic load balancing of read-only transactions between the Primary and Secondary servers. The default is "WRITE_MOSTLY" that corresponds to a normal HotStandby operation whereby all the load is transferred to the Primary server.
- "solid_tf1_reconnect_timeout"

  Set the timeout for reconnect retries, in the failure cases. The unit is millisecond. The default is 10000 (10 seconds).

For more information about using the TC connection properties, see the *IBM solidDB High Availability User Guide*.

Setting connection properties with the URL string

Any connection property can be also set, at connect time, within JDBC URL passed to the JDBC method DriverManager.getConnection(). In this case, the syntax of solidDB JDBC URL is the following:

```
"jdbc:solid://<hostname>:<port>/>username>/<password>[?<property-name>=<value>]..."
```

Examples

```
"jdbc:solid://localhost:1964/dba/dba"
"jdbc:solid://server1.acme.com:1964/dba/dba?solid_login_timeout_ms=100"
"jdbc:solid://server1.acme.com:1964/dba/dba?solid_login_timeout_ms=100?solid_idle_timeout_min=5"
```

JDBC 2.0 optional package API support

The solidDB JDBC Driver 2.0 supports selected features of the JDBC 2.0 specification Optional Package (known before as Standard Extension).

JDBC connection pooling

The JDBC 2.0 Standard Extension API specifies that users can implement a pooling technique by using specific caching or pooling algorithms that best suit their needs. A JDBC driver vendor must provide classes that implement the standard ConnectionPoolDataSource and PooledConnection interfaces

solidDB implements these classes as follows:

- ConnectionPoolDataSource

  A javax.sql.ConnectionPoolDataSource interface serves as a resource manager connection factory for pooled java.sql.Connection objects. solidDB provides the implementation for that interface in class SolidConnectionPoolDataSource. For API functions, see "ConnectionPoolDataSource API Functions" on page 69.

- PooledConnection

  A javax.sql.PooledConnection interface encapsulates the physical connection to a database. solidDB provides the implementation for that interface in class SolidPooledConnection. For API functions, see "PooledConnection API Functions" on page 75.

Note: solidDB does not provide an implementation for the actual connection pool (i.e. the data structure and the logic to actually pool the PooledConnection instances). Users must implement their own connection pooling logic (that is, a class that actually pools the connections).

ConnectionPoolDataSource API Functions

The public class SolidConnectionPoolDataSource implements javax.sql.ConnectionPoolDataSource. The API functions for javax.sql.ConnectionPoolDataSource interface are described as follows:

Table 22. Constructor

Description type	Description
Function Name	Constructor
Function Type	solidDB proprietary API
Description	Initializes class variables
Parameters	None
Return value	None
Syntax and exceptions	public SolidConnectionPoolDataSource()

Table 23. Constructor

Description type	Description
Function Name	Constructor
Function Type	solidDB proprietary API
Description	Initializes class variables
Parameters	url As String which identifies the DB server
Return value	None
Syntax and exceptions	public SolidConnectionPoolDataSource(String urlStr)

Table 24. setDescription

Description type	Description
Function Name	setDescription
Function Type	solidDB proprietary API
Description	This function sets the description string.
Parameters	description string (descString)
Return value	None
Syntax and exceptions	public void setDescription(String descString)

Table 25. getDescription

Description type	Description
Function Name	getDescription
Function Type	solidDB proprietary API
Description	This function returns the description string.
Parameters	None
Return value	returns a String (description)
Syntax and exceptions	public String getDescription()

Table 26. setURL

Description type	Description
Function Name	setURL
Function Type	solidDB proprietary API
Description	This function sets the url string which points to an solidDB server.
Parameters	url string (urlStr)
Return value	None
Syntax and exceptions	public void setURL(String urlStr)

Table 27. getURL

Description type	Description
Function Name	getURL
Function Type	solidDB proprietary API
Description	This function returns the DB url string.
Parameters	None
Return value	returns a String (url)
Syntax and exceptions	public String getURL()

Table 28. setUser

Description type	Description
Function Name	setUser
Function Type	solidDB proprietary API
Description	This function sets the username string. (WebSphere compatibility)
Parameters	username string
Return value	None
Syntax and exceptions	public void setUser(String newUser)

Table 29. getUser

Description type	Description
Function Name	getUser
Function Type	solidDB proprietary API
Description	This function returns the username string. (WebSphere compatibility)
Parameters	None
Return value	returns a String (username)
Syntax and exceptions	public String getUser()

Table 30. setPassword

Description type	Description
Function Name	setPassword
Function Type	solidDB proprietary API
Description	This function sets the password string. (WebSphere compatibility)
Parameters	password string
Return value	None
Syntax and exceptions	public void setPassword(String newPassword)

Table 31. getPassword

Description type	Description
Function Name	getPassword
Function Type	solidDB proprietary API
Description	This function returns the password string. (WebSphere compatibility)
Parameters	None
Return value	returns a String (password)
Syntax and exceptions	public String getPassword()

Table 32. setConnectionURL

Description type	Description
Function Name	setConnectionURL
Function Type	solidDB proprietary API
Description	This function sets the url string which points to an solidDB server.
Parameters	url string
Return value	None
Syntax and exceptions	public void setConnectionURL(String newUrl)

Table 33. getConnectionURL

Description type	Description
Function Name	getConnectionURL
Function Type	solidDB proprietary API
Description	This function returns the url string.
Parameters	None
Return value	returns a String (url)
Syntax and exceptions	public String getConnectionURL()

Table 34. getLoginTimeout

Description type	Description
Function Name	getLoginTimeout
Function Type	javax.sql.ConnectionPoolDataSource API
Description	This function returns the login timeout value.
Parameters	None
Return value	returns a timeout value as an integer (seconds)
Syntax and exceptions	public int getLoginTimeout() throws java.sql.SQLException

Table 35. getLogWriter

Description type	Description
Function Name	getLogWriter
Function Type	javax.sql.ConnectionPoolDataSource API
Description	This function returns the handle to a writer used for printing debugging messages.
Parameters	None
Return value	returns a handle to java.io.PrintWriter
Syntax and exceptions	public java.io.PrintWriter getLogWriter() throws java.sql.SQLException

Table 36. getPooledConnection

Description type	Description
Function Name	getPooledConnection
Function Type	javax.sql.ConnectionPoolDataSource API
Description	This function returns a PooledConnection object from the connection pool. This object has a valid connection to the database server.
Parameters	None
Return value	returns a PooledConnection object.
Syntax and exceptions	public javax.sql.PooledConnection getPooledConnection() throws java.sql.SQLException

Table 37. getPooledConnection

Description type	Description
Function Name	getPooledConnection
Function Type	javax.sql.ConnectionPoolDataSource API
Description	This function returns a PooledConnection object from the connection pool. This object has a valid connection to the database server.
Parameters	user (username as String), password (password as String)
Return value	returns a PooledConnection object.
Syntax and exceptions	public javax.sql.PooledConnection getPooledConnection(String user, String password) throws java.sql.SQLException

Table 38. setLoginTimeout

Description type	Description
Function Name	setLoginTimeout
Function Type	javax.sql.ConnectionPoolDataSource API
Description	This function sets the login timeout value in seconds
Parameters	seconds (as integer)
Return value	None

Table 38. setLoginTimeout (continued)

Description type	Description
Syntax and exceptions	public void setLoginTimeout(int seconds)

Table 39. setLogWriter

Description type	Description
Function Name	setLogWriter
Function Type	javax.sql.ConnectionPoolDataSource API
Description	This function sets the handle to a writer object that will be used to print/log debug messages.
Parameters	handle to java.io.PrintWriter
Return value	None
Syntax and exceptions	public void setLogWriter(java.io.PrintWriter out) throws java.sql.SQLException

PooledConnection API Functions

The public class SolidPooledConnection implements javax.sql.PooledConnection. The API functions for javax.sql.PooledConnection interface are:

Table 40. addConnectionEventListener

Description type	Description
Function Name	addConnectionEventListener
Function Type	javax.sql.PooledConnection API
Description	Adds an event listener to whom this object should notify when it wants to release the connection. This listener is generally the connection pooling module.
Parameters	listener (handle to javax.sql.ConnectionEventListener)
Return value	None
Syntax and exceptions	public void addConnectionEventListener(javax.sql.ConnectionEventListener listener)

Table 41. close

Description type	Description
Function Name	close
Function Type	javax.sql.PooledConnection API
Description	This function closes the physical connection.
Parameters	None
Return value	None
Syntax and exceptions	public void close() throws java.sql.SQLException

Table 42. getConnection

Description type	Description
Function Name	getConnection
Function Type	javax.sql.PooledConnection API
Description	returns a handle to java.sql.Connection
Parameters	None
Return value	java.sql.Connection
Syntax and exceptions	public java.sql.Connection getConnection() throws java.sql.SQLException

Table 43. removeConnectionEventListener

Description type	Description
Function Name	removeConnectionEventListener
Function Type	javax.sql.PooledConnection API
Description	This function removes the reference to the listener
Parameters	listener
Return value	None
Syntax and exceptions	public void removeConnectionEventListener(javax.sql.ConnectionEventListener listener)

solidDB Connected RowSet Class: SolidJDBCRowSet

The RowSet described in this topic, extends solid.jdbc.SolidBaseRowSet (which implements javax.sql.RowSet) constructors.

```
/**
 * Create a SolidJDBCRowSet with an existing Connection handle  */
public SolidJDBCRowSet(java.sql.Connection conn)

/**
 * Create a SolidJDBCRowSet with an existing ResultSet handle  */
public SolidJDBCRowSet(java.sql.ResultSet rset)

/**
 * Create a new SolidJDBCRowSet with given url, username and
 * password.
 */
public SolidJDBCRowSet(String url, String uname, String pwd)

/**
 * Create a new SolidJDBCRowSet with given url, username,
 * password and JNDI naming context.
 */
public SolidJDBCRowSet(String dsname,
                       String username,
                       String password,
                       Context namingcontext)
```

Ssee the method interface in, for example: http://java.sun.com/j2se/1.4.2/docs/api/javax/sql/RowSet.html

Considerations about the Usage of SolidJDBCRowSet

There are certain methods that you can call (usually for setting parameters for commands to be executed or setting the properties of the RowSet instance) before a connection to the database has been made. However, most of the RowSet interface methods can be called only after a connection to the database has been made. This means that method a command has been set with method setCommand(String) and method execute() has been called. If the SolidJDBCRowSet instance has no previous java.sql.Connection handle, the connection will be established during execute() call. After the execute() call, the row set instance contains a java.sql.Connection object, a java.sql.PreparedStatement object, and if the command execute was a query statement, it contains also a java.sql.ResultSet handle. It also contains all parameter setting methods: setString, setObject, and so on.

The following example describes the proper use of SolidJDBCRowSet class.

```
/**
 * A simple example on how to use SolidJDBCRowSet
 * First: create  an instance of a connected RowSet class.
 * Naturally, you can give the url, username and password
 * right away in the constructor below, but null parameters
 * for the corresponding values have bee given in the example
 * just to show how to use setUrl, setUsername etc. methods of the
 * RowSet class.
 */
SolidJDBCRowSet rs = new SolidJDBCRowSet(null, null, null);

// Set the url for the connection
rs.setUrl("jdbc:solid://localhost:1313");

// set the username
rs.setUsername("user1");
```

```
// set the passwd
rs.setPassword("pwd1");


/**
 * Note! You can set command parameters and other properties
 * in any order you like, for example, you can set the parameters
 * before you have defined the command to be executed. You can
 * also define the command parameters in any order, since the
 * command statement as well as the given parameters won't be
 * parsed until a connection to the database has been made in
 * the execute() method call.
 */

// set parameter #2 for the command
rs.setString(2,"'SYS_SYNC%'");

// set the command string
rs.setCommand("select table_name from tables where table_name like ?
and table_name not like ?;");

// set the parameter #1
rs.setString(1,"'SYS_%'");

// execute the command. The connection to the database is not
// established before this call.
rs.execute();

// now you can browse the ResultSet
while( rowset.next() ){
   // do stuff
}

// close the result set. This method call closes the connection
// to the database as well.
rs.close()
```

Java Naming and Directory Interface (JNDI)

The solidDB JDBC 2.0 Driver supports the Java Naming and Directory Interface (JNDI).

JNDI allows applications to access naming and directory services through a common interface. JNDI is not a service, but a set of interfaces. These interfaces allow applications to access many different directory services including: file systems, directory services such as Lightweight Directory Access Protocol (LDAP), Network Information System (NIS), and distributed object systems such as the Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), and Enterprise JavaBeans™ (EJB).

Code examples

This topic contains four Java code samples that use the solidDB JDBC driver.

Java Code Example 1

```
/**
 *      sample1 JDBC sample application
 *
 *
 *      This simple JDBC application does the following using
 *      Solid JDBC driver.
 *
 *  1. Registers the driver using JDBC driver manager services
```

```
 *  2. Prompts the user for a valid JDBC connect string
 *  3. Connects to Solid using the driver
 *  4. Creates a statement for one query,
 *     'SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM TABLES'
 *     for reading data from one of the Solid system tables.
 *  5. Executes the query
 *  6. Fetches and dumps all the rows of a result set.
 *  7. Closes connection
 *
 *  To build and run the application
 *
 *  1. Make sure you have a working Java Development environment
 *  2. Install and start Solid to connect. Ensure that the
 *     server is up and running.
 *  3. Append SolidDriver2.0.jar into the CLASSPATH definition used
 *     by your development/running environment.
 *  4. Create a java project based on the file sample1.java.
 *  5. Build and run the application.
 *
 *  For more information read the readme.txt file contained in the
 *  solidDB package.
 *
 */

import java.io.*;

public class sample1 {

  public static void main (String args[]) throws Exception
  {
    java.sql.Connection conn;
    java.sql.ResultSetMetaData meta;
    java.sql.Statement stmt;
    java.sql.ResultSet result;
    int i;

    System.out.println("JDBC sample application starts...");
    System.out.println("Application tries to register the driver.");

    // this is the recommended way for registering Drivers
    java.sql.Driver d =
       (java.sql.Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();

    System.out.println("Driver succesfully registered.");

    // the user is asked for a connect string
    System.out.println(
        "Now sample application needs a connectstring in format:\n"
        );
    System.out.println(
        "jdbc:solid://<host>:<port>/<user name>/<password>\n"
        );
    System.out.print("\nPlease enter the connect string >");
    BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
    String sCon = reader.readLine();

    // next, the connection is attempted
    System.out.println("Attempting to connect :" + sCon);
    conn = java.sql.DriverManager.getConnection(sCon);

    System.out.println("SolidDriver succesfully connected.");

    String sQuery = "SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM TABLES";

    stmt= conn.createStatement();
```

```
    result = stmt.executeQuery(sQuery);
    System.out.println("Query executed and result set obtained.");

    // we get a metadataobject containing information about the
    // obtained result set
    System.out.println("Obtaining metadata information.");
    meta = result.getMetaData();
    int cols = meta.getColumnCount();

    System.out.println("Metadata information for columns is as follows:");
    // we dump the column information about the result set
    for (i=1; i <= cols; i++)
    {
        System.out.println("Column i:"+i+"  "+meta.getColumnName(i)+ "," +
             meta.getColumnType(i) + "," + meta.getColumnTypeName(i));
    }

    // and finally, we dump the result set
    System.out.println("Starting to dump result set.");
    int cnt = 1;
    while(result.next())
    {
        System.out.print("\nRow "+cnt+" : ");
        for (i=1; i <= cols; i++) {
            System.out.print(result.getString(i)+"\t");
        }
        cnt++;
    }

    stmt.close();

    conn.close();
    // and not it is all over
    System.out.println("\nResult set dumped. Sample application finishes.");
  }
}
```

Java Code Example 1 Output

```
Solid\DatabaseEngine4.1\jdbc\samples>java sample1.java
JDBC sample application starts...
Application tries to register the driver.
Driver succesfully registered.
Now sample application needs a connectstring in format:

jdbc:solid://<host>:<port>/<user name>/<password>

Please enter the connect string >jdbc:solid://localhost:1313/dba/dba
Attempting to connect :jdbc:solid://localhost:1313/dba/dba
SolidDriver succesfully connected.
Query executed and result set obtained.
Obtaining metadata information.
Metadata information for columns is as follows:
Column i:1  TABLE_SCHEMA,12,VARCHAR
Column i:2  TABLE_NAME,12,VARCHAR
Column i:3  TABLE_TYPE,12,VARCHAR
Starting to dump result set.

Row 1 : _SYSTEM SYS_TABLES      BASE TABLE
Row 2 : _SYSTEM SYS_COLUMNS     BASE TABLE
Row 3 : _SYSTEM SYS_USERS       BASE TABLE
Row 4 : _SYSTEM SYS_UROLE       BASE TABLE
Row 5 : _SYSTEM SYS_RELAUTH     BASE TABLE
Row 6 : _SYSTEM SYS_ATTAUTH     BASE TABLE
Row 7 : _SYSTEM SYS_VIEWS       BASE TABLE
Row 8 : _SYSTEM SYS_KEYPARTS    BASE TABLE
Row 9 : _SYSTEM SYS_KEYS        BASE TABLE
```

```
Row 10 : _SYSTEM        SYS_CARDINAL    BASE TABLE
Row 11 : _SYSTEM        SYS_INFO        BASE TABLE
Row 12 : _SYSTEM        SYS_SYNONYM     BASE TABLE
Row 13 : _SYSTEM        TABLES  VIEW
Row 14 : _SYSTEM        COLUMNS VIEW
Row 15 : _SYSTEM        SQL_LANGUAGES   BASE TABLE
Row 16 : _SYSTEM        SERVER_INFO     VIEW
Row 17 : _SYSTEM        SYS_TYPES       BASE TABLE
Row 18 : _SYSTEM        SYS_FORKEYS     BASE TABLE
Row 19 : _SYSTEM        SYS_FORKEYPARTS BASE TABLE
Row 20 : _SYSTEM        SYS_PROCEDURES  BASE TABLE
Row 21 : _SYSTEM        SYS_TABLEMODES  BASE TABLE
Row 22 : _SYSTEM        SYS_EVENTS      BASE TABLE
Row 23 : _SYSTEM        SYS_SEQUENCES   BASE TABLE
Row 24 : _SYSTEM        SYS_TMP_HOTSTANDBY      BASE TABLE
Result set dumped. Sample application finishes.
```

Java Code Example 2

```
/**
 *      sample2 JDBC sample applet
 *
 *
 *      This simple JDBC applet does the following using
 *      Solid native JDBC driver.
 *
 *  1. Registers the driver using JDBC driver manager services
 *  2. Connects to Solid using the driver.
 *     Used url is read from sample2.html
 *  3. Executes given SQL statements
 *
 *  To build and run the application
 *
 *  1. Make sure you have a working Java Development environment
 *  2. Install and start Solid to connect. Ensure that
 *  the server is up and running.
 *  3. Append SolidDriver2.0.jar into the CLASSPATH definition used
 *     by your development/running environment.
 *  4. Create a java project based on the file sample2.java.
 *  5. Build and run the application. Check that sample2.html
 *     defines valid url to your environment.
 *
 *  For more information read the readme.txt file contained
 *  in the IBM SolidDB Development Kit package.
*
 */

import java.util.*;
import java.awt.*;
import java.applet.Applet;
import java.net.URL;
import java.sql.*;

public class sample2 extends Applet {
  TextField textField;
  static TextArea textArea;

  String url = null;
  Connection con = null;

  public void init() {
      // a valid value for url could be
      // url = "jdbc:solid://localhost:1313/dba/dba";

      url = getParameter("url");

      textField = new TextField(40);
```

```
        textArea = new TextArea(10, 40);
        textArea.setEditable(false);

        Font font = textArea.getFont();
        Font newfont = new Font("Monospaced", font.PLAIN, 12);
        textArea.setFont(newfont);

        // Add Components to the Applet.
        GridBagLayout gridBag = new GridBagLayout();
        setLayout(gridBag);
        GridBagConstraints c = new GridBagConstraints();
        c.gridwidth = GridBagConstraints.REMAINDER;

        c.fill = GridBagConstraints.HORIZONTAL;
        gridBag.setConstraints(textField, c);
        add(textField);

        c.fill = GridBagConstraints.BOTH;
        c.weightx = 1.0;
        c.weighty = 1.0;
        gridBag.setConstraints(textArea, c);
        add(textArea);

        validate();

        try {
            // Load the Solid JDBC Driver
            Driver d =
               (Driver)Class.forName ("solid.jdbc.SolidDriver").newInstance();

            // Attempt to connect to a driver.
            con = DriverManager.getConnection (url);

            // If we were unable to connect, an exception
            // would have been thrown.  So, if we get here,
            // we are successfully connected to the url

            // Check for, and display and warnings generated
            // by the connect.
            checkForWarning (con.getWarnings ());

            // Get the DatabaseMetaData object and display
            // some information about the connection
            DatabaseMetaData dma = con.getMetaData ();

            textArea.appendText("Connected to " + dma.getURL() + "\n");
            textArea.appendText("Driver       " + dma.getDriverName() + "\n");
            textArea.appendText("Version      " + dma.getDriverVersion() + "\n");
        }
        catch (SQLException ex) {
            printSQLException(ex);
        }
        catch (Exception e) {
            textArea.appendText("Exception:  " + e + "\n");
        }
    }

    public void destroy() {
        if (con != null) {
            try {
                con.close();
            }
            catch (SQLException ex) {
                printSQLException(ex);
            }
            catch (Exception e) {
                textArea.appendText("Exception:  " + e + "\n");
```

```
        }
    }
}

public boolean action(Event evt, Object arg) {
    if (con != null) {
        String sqlstmt = textField.getText();
        textArea.setText("");
        try {
            // Create a Statement object so we can submit
            // SQL statements to the driver
            Statement stmt = con.createStatement ();
            // set row limit
            stmt.setMaxRows(50);
            // Submit a query, creating a ResultSet object
            ResultSet rs = stmt.executeQuery (sqlstmt);

            // Display all columns and rows from the result set
            textArea.setVisible(false);
            dispResultSet (stmt,rs);
            textArea.setVisible(true);

            // Close the result set
            rs.close();

            // Close the statement
            stmt.close();
        }
        catch (SQLException ex) {
            printSQLException(ex);
        }
        catch (Exception e) {
            textArea.appendText("Exception:  " + e + "\n");
        }
        textField.selectAll();
    }
    return true;
}

  //-------------------------------------------------------------------
  // checkForWarning
  // Checks for and displays warnings.  Returns true if a warning
  // existed
  //-------------------------------------------------------------------

  private static boolean checkForWarning (SQLWarning warn)
          throws SQLException
  {
      boolean rc = false;

      // If a SQLWarning object was given, display the
      // warning messages.  Note that there could be
      // multiple warnings chained together

      if (warn != null) {
          textArea.appendText("\n*** Warning ***\n");
          rc = true;
          while (warn != null) {
              textArea.appendText("SQLState: " +
                  warn.getSQLState () + "\n");
              textArea.appendText("Message:  " +
                  warn.getMessage () + "\n");
              textArea.appendText("Vendor:   " +
                  warn.getErrorCode () + "\n");
              textArea.appendText("\n");
              warn = warn.getNextWarning ();
          }
```

```
    }
    return rc;
}

//----------------------------------------------------------------
// dispResultSet
// Displays all columns and rows in the given result set
//----------------------------------------------------------------

private static void dispResultSet (Statement sta, ResultSet rs)
    throws SQLException
{
    int i;

    // Get the ResultSetMetaData.  This will be used for
    // the column headings
    ResultSetMetaData rsmd = rs.getMetaData ();

    // Get the number of columns in the result set
    int numCols = rsmd.getColumnCount ();
    if (numCols == 0) {
        textArea.appendText("Updatecount is "+sta.getUpdateCount());
        return;
    }

    // Display column headings
    for (i=1; i<=numCols; i++) {
        if (i > 1) {
            textArea.appendText("\t");
        }
        try {
            textArea.appendText(rsmd.getColumnLabel(i));
        }
        catch(NullPointerException ex) {
            textArea.appendText("null");
        }
    }
    textArea.appendText("\n");

    // Display data, fetching until end of the result set
    boolean more = rs.next ();
    while (more) {

        // Loop through each column, get the
        // column datza and display it
        for (i=1; i<=numCols; i++) {
            if (i > 1) {
                textArea.appendText("\t");
            }
            try {
                textArea.appendText(rs.getString(i));
            }
            catch(NullPointerException ex) {
                textArea.appendText("null");
            }
        }
        textArea.appendText("\n");

        // Fetch the next result set row
        more = rs.next ();
    }
}

private static void printSQLException(SQLException ex)
{
        // A SQLException was generated.  Catch it and
        // display the error information.  Note that there
```

```
            // could be multiple error objects chained
            // together

            textArea.appendText("\n*** SQLException caught ***\n");

            while (ex != null) {
                textArea.appendText("SQLState: " +
                    ex.getSQLState () + "\n");
                textArea.appendText("Message:  " +
                    ex.getMessage () + "\n");
                textArea.appendText("Vendor:   " +
                    ex.getErrorCode () + "\n");
                textArea.appendText("\n");
                ex = ex.getNextException ();
            }
    }
}
```

Java Code Example 3

```
/**
 *      sample3 JDBC sample application
 *
 *
 *      This simple JDBC application does the following using
 *      Solid JDBC driver.
 *
 *  1. Registers the driver using JDBC driver manager services
 *  2. Prompts the user for a valid JDBC connect string
 *  3. Connects to Solid using the driver
 *  4. Drops and creates a procedure sample3. If the procedure
 *     does not exist dumps the related exception.
 *  5. Calls that procedure using java.sql.Statement
 *  6. Fetches and dumps all the rows of a result set.
 *  7. Closes connection
 *
 *  To build and run the application
 *
 *  1. Make sure you have a working Java Development environment
 *  2. Install and start Solid to connect. Ensure that the
 *     server is up and running.
 *  3. Append SolidDriver2.0.jar into the CLASSPATH definition used
 *     by your development/running environment.
 *  4. Create a java project based on the file sample3.java.
 *  5. Build and run the application.
 *
*  For more information read the readme.txt
*  file contained in the solidDB Development Kit package.
*/

import java.io.*;
import java.sql.*;

public class sample3 {

    static Connection conn;
    public static void main (String args[]) throws Exception
    {
        System.out.println("JDBC sample application starts...");
        System.out.println("Application tries to register the driver.");

        // this is the recommended way for registering Drivers
        Driver d = (Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();

        System.out.println("Driver succesfully registered.");

        // the user is asked for a connect string
```

```
    System.out.println(
        "Now sample application needs a connectstring in format:\n"
        );
    System.out.println(
        "jdbc:solid://<host>:<port>/<user name>/<password>\n"
        );
    System.out.print("\nPlease enter the connect string >");
    BufferedReader reader =
    new BufferedReader(new InputStreamReader(System.in));
    String sCon = reader.readLine();

    // next, the connection is attempted
    System.out.println("Attempting to connect :" + sCon);
    conn = DriverManager.getConnection(sCon);

    System.out.println("SolidDriver succesfully connected.");

    DoIt();

    conn.close();
    // and now it is all over
    System.out.println(
        "\nResult set dumped. Sample application finishes."
        );
}


static void DoIt() {
    try {
        createprocs();
        PreparedStatement pstmt = conn.prepareStatement("call sample3(?)");
        // set parameter value
        pstmt.setInt(1,10);

        ResultSet rs = pstmt.executeQuery();
        if (rs != null) {
            ResultSetMetaData md = rs.getMetaData();
            int cols = md.getColumnCount();
            int row = 0;
            while (rs.next()) {
                row++;
                String ret = "row "+row+": ";
                for (int i=1;i<=cols;i++) {
                    ret = ret + rs.getString(i) + " ";
                }
                System.out.println(ret);
            }
        }
        conn.commit();
    }
    catch (SQLException ex) {
        printexp(ex);
    }
    catch (java.lang.Exception ex) {
        ex.printStackTrace ();
    }

}

static void createprocs() {
    Statement stmt = null;
    String proc = "create procedure sample3 (limit integer)" +
                  "returns (c1 integer, c2 integer) " +
                  "begin " +
                  "  c1 := 0;" +
                  "  while c1 < limit loop " +
                  "    c2 := 5 * c1;" +
```

```
                      "    return row;" +
                      "    c1 := c1 + 1;" +
                      "  end loop;" +
                      "end";

        try {
            stmt = conn.createStatement();
            stmt.execute("drop procedure sample3");
        } catch (SQLException ex) {
            printexp(ex);
        }

        try {
            stmt.execute(proc);
        } catch (SQLException ex) {
            printexp(ex);
            System.exit(-1);
        }
    }

    public static void printexp(SQLException ex) {
        System.out.println("\n*** SQLException caught ***");
        while (ex != null) {
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("Message:  " + ex.getMessage());
            System.out.println("Vendor:   " + ex.getErrorCode());
            ex = ex.getNextException ();
        }
    }

}
```

Java Code Example 4

```
/**
 *      sample4 JDBC sample application
 *
 *
 *      This simple JDBC application does the following using
 *      Solid JDBC driver.
 *
 *  1. Registers the driver using JDBC driver manager services
 *  2. Prompts the user for a valid JDBC connect string
 *  3. Connects to Solid using the driver
 *  4. Drops and creates a table sample4. If the table
 *     does not exist dumps the related exception.
 *  5. Inserts file given as an argument to database (method Store)
 *  6. Reads this 'blob' back to file out.tmp (method Restore)
 *  7. Closes connection
 *
 *  To build and run the application
 *
 *  1. Make sure you have a working Java Development environment
 *  2. Install and start Solid to connect. Ensure that
 *   the server is up and running.
 *  3. Append SolidDriver2.0.jar into the CLASSPATH definition used
 *     by your development/running environment.
 *  4. Create a java project based on the file sample4.java.
 *  5. Build and run the application.
 *
 *  For more information read the readme.txt file
 *  contained in the solidDB Development Kit package.
*
 */

import java.io.*;
import java.sql.*;
```

```
public class sample4 {

    static Connection conn;
    public static void main (String args[]) throws Exception
    {
        String filename = null;
        String tmpfilename = null;

        if (args.length < 1) {
            System.out.println("usage: java sample4 <infile>");
            System.exit(0);
        }
        filename = args[0];
        tmpfilename = "out.tmp";
        System.out.println("JDBC sample application starts...");
        System.out.println("Application tries to register the driver.");

        // this is the recommended way for registering Drivers
        Driver d = (Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();

        System.out.println("Driver succesfully registered.");

        // the user is asked for a connect string
        System.out.println(
            "Now sample application needs a connectstring in format:\n"
            );
        System.out.println(
            "jdbc:solid://<host>:<port>/<user name>/<password>\n"
            );
        System.out.print("\nPlease enter the connect string >");
        BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
        String sCon = reader.readLine();

        // next, the connection is attempted
        System.out.println("Attempting to connect :" + sCon);
        conn = DriverManager.getConnection(sCon);

        System.out.println("SolidDriver succesfully connected.");

        // drop and create table sample4
        createsample4();
        // insert data into it
        Store(filename);
        // and restore it
        Restore(tmpfilename);

        conn.close();
        // and it is all over
        System.out.println("\nSample application finishes.");
    }


    static void Store(String filename) {
        String sql = "insert into sample4 values(?,?)";
        FileInputStream inFileStream ;
        try {
            File f1 = new File(filename);
            int blobsize = (int)f1.length();
            System.out.println("Inputfile size is "+blobsize);
            inFileStream = new FileInputStream(f1);

            PreparedStatement stmt = conn.prepareStatement(sql);
            stmt.setLong(1, System.currentTimeMillis());
            stmt.setBinaryStream(2, inFileStream, blobsize);
            int rows = stmt.executeUpdate();
```

```
            stmt.close();
            System.out.println(""+rows+" inserted.");
            conn.commit();
        }
        catch (SQLException ex) {
            printexp(ex);
        }
        catch (java.lang.Exception ex) {
            ex.printStackTrace ();
        }

    }

    static void Restore(String filename) {
        String sql = "select id,blob from sample4";
        FileOutputStream outFileStream ;
        try {
            File f1 = new File(filename);
            outFileStream = new FileOutputStream(f1);

            PreparedStatement stmt = conn.prepareStatement(sql);
            ResultSet rs = stmt.executeQuery();
            int readsize = 0;
            while (rs.next()) {
                InputStream in = rs.getBinaryStream(2);
                byte bytes[] = new byte[8*1024];
                int nRead = in.read(bytes);
                while (nRead != -1) {
                    readsize = readsize + nRead;
                    outFileStream.write(bytes,0,nRead);
                    nRead = in.read(bytes);
                }

            }
            stmt.close();
            System.out.println("Read "+readsize+" bytes from database");
        }
        catch (SQLException ex) {
            printexp(ex);
        }
        catch (java.lang.Exception ex) {
            ex.printStackTrace ();
        }

    }


    static void createsample4() {
        Statement stmt = null;
        String proc = "create table sample4 (" +
                      "id numeric not null primary key,"+
                      "blob long varbinary)";

        try {
            stmt = conn.createStatement();
            stmt.execute("drop table sample4");
        } catch (SQLException ex) {
            printexp(ex);
        }

        try {
            stmt.execute(proc);
        } catch (SQLException ex) {
            printexp(ex);
            System.exit(-1);
        }
    }
```

```
    static void printexp(SQLException ex) {
        System.out.println("\n*** SQLException caught ***");
        while (ex != null) {
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("Message:  " + ex.getMessage());
            System.out.println("Vendor:   " + ex.getErrorCode());
            ex = ex.getNextException ();
        }
    }

}
```

solidDB JDBC Driver type conversion matrix

The conversion matrix included in this topic shows how the Java data type to SQL data type conversion is supported by solidDB JDBC Driver.

This matrix applies to both ResultSet.getXXX and ResultSet.setXXX methods for getting and setting data. An X indicates that the method is supported by solidDB JDBC Driver.

Table 44. Java data type to SQL data type conversion

Java Data Type **applies to getting and setting data**	**TINYINT**	**SMALLINT**	**INTEGER**	**REAL**	**FLOAT**	**DOUBLE**	**DECIMAL**	**NUMERIC**	**CHAR**	**VARCHAR**	**LONGVARCHAR**	**WCHAR**	**WVARCHAR**	**LONGWVARCHAR**	**BINARY**	**VARBINARY**	**LONGVARBINARY**	**DATE**	**TIME**	**TIMESTAMP**
getArray/setArray																				
getBlob/setBlob																				
getByte/setByte	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
getCharacterStream/ setCharacterStream									X	X	X	X	X	X	X	X	X	X	X	X
getClob/setClob																				
getShort/setShort	X	X	X	X	X	X	X	X	X	X	X									
getInt/setInt	X	X	X	X	X	X	X	X	X	X	X									
getlong/setLong	X	X	X	X	X	X	X	X	X	X	X									
getfloat/setfloat	X	X	X	X	X	X	X	X	X	X	X									
getDouble/setDouble	X	X	X	X	X	X	X	X	X	X	X									
getBigDecimal/setBigDecimal	X	X	X	X	X	X	X	X	X	X	X									
getRef/setRef																				

Table 44. Java data type to SQL data type conversion (continued)

Java Data Type **applies to getting and setting data**	**TINYINT**	**SMALLINT**	**INTEGER**	**REAL**	**FLOAT**	**DOUBLE**	**DECIMAL**	**NUMERIC**	**CHAR**	**VARCHAR**	**LONGVARCHAR**	**WCHAR**	**WVARCHAR**	**LONGWVARCHAR**	**BINARY**	**VARBINARY**	**LONGVARBINARY**	**DATE**	**TIME**	**TIMESTAMP**
getBoolean/setBoolean	X	X	X	X	X	X	X	X	X	X	X									
getString/setString	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
getBytes/setBytes									X	X	X	X	X	X	X	X	X			
getDate/setDate									X	X	X	X	X	X				X		X
getTime/setTime									X	X	X	X	X	X					X	X
getTimestamp/setTimestamp									X	X	X	X	X	X				X		X
getAsciiStream/setAsciiStream									X	X	X	X	X	X	X	X	X			
getUnicodeStream/setUnicodeStream									X	X	X	X	X	X	X	X	X			
getBinaryStream/setBinaryStream									X	X	X	X	X	X	X	X	X			
getObject/setObject	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

5 Using solidDB SA

This section describes how to use the solidDB Application Programming Interface (API) also known as *solidDB SA*

solidDB SA is a low level C-language client library to access solidDB database management products. solidDB SA is a layer that resides internally in solidDB products. Normally, the use of an industry standards based interface, such as ODBC or JDBC, is recommended. However, in environments with heavy write load (BATCH INSERTS AND UPDATES), solidDB SA can provide a significant performance advantage.

What Is solidDB SA?

solidDB SA is a C-language client library to connect solidDB database products. This library is used internally in solidDB products and provides access to data in solidDB database tables. The library contains 90 functions providing low-level mechanisms for connecting the database and running cursor-based operations.

Compared to industry standard interfaces, such as ODBC or JDBC, solidDB SA offers better flexibility in constructing network messages that are sent to the database server. In applications whose read or write performance (for example, batch inserts or primary key lookups) needs to be optimized, using solidDB SA can provide a significant performance advantage. For example, if your site experiences performance bottlenecks during batch inserts, solidDB SA can reduce the bottleneck because solidDB SA lets you pass several rows for insertion inside a single network message or remote procedure call.

Note: If the performance bottleneck is in read operations, using IBM solidDB SA will provide only minor improvement in performance.

The solidDB SA interface bypasses the SQL parser, interpreter, and optimizer. With solidDB SA you can access result sets, as long as you are not using SQL through solidDB SA. If retrieval of result sets is necessary through SQL, then you must use industry standard APIs such as ODBC or the solidDB Light Client based on ODBC.

To use solidDB SA requires that you convert your existing interface. This is why we recommend that you use solidDB SA only after you have already attempted (with little success) to use other means for improving performance, which include:

- Writing or indexing columns by primary key for the most appropriate row order. solidDB, otherwise, stores rows on disk in the order they are inserted into the database.
- Eliminating unnecessary indexes. For example, a query that selects more than 15% of a table's rows may be performed faster by a full table scan.
- Optimizing the transaction size by committing transactions after every 100-200 rows inserted.
- Using stored procedures.

Getting started with solidDB SA

This topic describes the steps that you should take before using solidDB SA.

Before getting started with solidDB SA, be sure you have:

1. Obtained the solidDB with linked library access library. This library includes:
   - Linkable libraries that provide local solidDB SA and server functionality
   - Sample solidDB SA files to help you get started. These files include a ready-to-link C sample application, along with a readme text file (`readme.txt`).
2. If you are building a remote user application, you need the following library so that you can link it into your application: `solidimpsa.lib`. Note that the filename extension may vary by platform.
3. Started solidDB. If necessary, create a new database before using solidDB SA.

Setting up the Development Environment and Building a Sample Program

Building an application program using the solidDB SA library in the linked library access or the SA client library is identical to building any normal C/C++ program:

1. Insert the linked library access library file or SA client library into your project. Refer to section "Linking applications for the linked library access" in the *IBM solidDB Linked Library Access User Guide* for the correct filenames.
2. Include the following solidDB SA header file, which is required in applications using solidDB SA library in the linked library access or the solidDB SA client library:

   ```
   #include "sa.h"
   ```

   Insert the directory containing all the other necessary solidDB SA headers into your development environment's include directories setting.
3. Compile the source code.
4. Link the program.

Verifying the Development Environment Setup

The easiest way to verify the development setup is to use a solidDB SA sample program. This enables you to verify your development environment without writing any code.

Verify the following items in your development environment:

- In the Windows environment, the TCP/IP services are provided by standard DLL `wsock32.dll`. To link these services into your project, add `wsock32.lib` into linker's lib file list.
- On VxWorks target machines, you should run a kernel that has a working TCP/IP stack running. Usually you can verify this by checking that the target machine responds to ping requests. For example, if you have configured your target machine to have an IP address 192.168.1.111, you would run ping 192.168.1.111 from another workstation in your LAN for a response that proves that the target is active:

  ```
  C:\>ping 192.168.1.111
  Pinging 192.168.1.111 with 32 bytes of data:
  Reply from 192.168.1.111: bytes=32 time=260ms TTL=62
  ```

  After verification, your solidDB SA application should work on that target machine.

Connecting to a Database by Using the Sample Application

In solidDB SA, a connection to a database is represented by the SaConnectT structure. This structure is established by calling the function SaConnect. The following sample code establishes a connection to a database listening TCP/IP protocol at local machine port 1313. User account DBA with password DBA has been defined in the database.

```
SaConnectT* scon;

scon = SaConnect("tcp localhost 1313", "dba", "dba");
if (scon == NULL)
{
    /* Connect failed, display connect error text. */
    char* errstr;
    SaErrorInfo(NULL, &errstr, NULL);
    printf("%s\n", errstr);
    return(1);
}
```

Writing data by using solidDB SA without SQL

With solidDB SA, data is written using cursors.

For delete and update operations, after the cursor is created, a search is performed so that the cursor points to the row that is to be updated and deleted. For insert operations, after the cursor is created, the insertion row(s) are immediately written to the cursor. solidDB SA also enables passing several rows for insertion inside a single network message.

Performing insert operations

solidDB SA functions required for insert operations are listed in the table below.

After solidDB creates a cursor to a certain table, variables are bound to columns, rows are written to the cursor, and then the cursor is closed.

Note: If you use SaArrayInsert to insert more than one row in a single message, then you must perform an explicit flush to send the rows to the database.

Table 45. Insert operation steps

Steps	SA Function(s)	Comment
1. Create a cursor	SaCursorCreate	
2. Binding variables to cursor	SaCursorColData, SaCursorColDate, SaCursorColDateFormat, SaCursorColDFloat, SaCursorColDouble, SaCursorColDynData, SaCursorColDynstr, SaCursorColFloat, SaCursorColInt, SaCursorColLong, SaCursorColStr, SaCursorColTime, SaCursorColTimestamp	
3. Open the cursor	SaCursorOpen	
4. Write a row(s) to the cursor	SaArrayInsert for more than one row or SaCursorInsert for a single row	Perform this in a loop if necessary

Table 45. Insert operation steps (continued)

Steps	SA Function(s)	Comment
5. Free the cursor	SaCursorFree	
6. Flush the network message to the server	SaArrayFlush	Necessary only if using SaArrayInsert.

The following code sample excerpt demonstrates how to write four rows of data in a single network message using the SaArrayInsert function. In the code, a call to SaArrayFlush flushes all rows to the server so they are passed in the same network message.

```
scur = SaCursorCreate(scon, "SAEXAMPLE");

/* Bind variables to columns. */
SaCursorColInt(scur, "INTC", &intc);
SaCursorColStr(scur, "CHARC", &str);

/* Open the cursor. */
SaCursorOpen(scur);

/* Insert values to the table. The column values are taken
 * from the user variables that are bound to columns.
 */
for (intc = 2; intc <= 5; intc++) {
    switch (intc) {
        case 2:
            str = "B";
            break;
        case 3:
            str = "C";
            break;
        case 4:
            str = "D";
            break;
        case 5:
            str = "E";
            break;
    }
    SaArrayInsert(scur);
}

/* Close the cursor. */
SaCursorFree(scur);

/* Flush the inserts to the server. */
SaArrayFlush(scon, NULL);
```

Performing update and delete operations

solidDB SA functions required for basic update and delete operations are listed in the table below.

After solidDB creates a cursor to a specific table, variables are bound to columns of the table, and the cursor is opened. Before the actual search begins, the constraints for finding the row for deletion are set. If there are more rows to be updated, each of the rows requires a separate fetch before they are updated or deleted. After the operation, the cursor is freed.

Table 46. Update and delete operation steps

Steps	SA Function(s)	Comment
1. Create a cursor	SaCursorCreate	
2. Binding variables to cursor	SaCursorColData, SaCursorColDate, SaCursorColDateFormat, SaCursorColDFloat, SaCursorColDouble, SaCursorColDynData, SaCursorColDynstr, SaCursorColFloat, SaCursorColInt, SaCursorColLong, SaCursorColStr, SaCursorColTime, SaCursorColTimestamp	
3. Open the cursor	SaCursorOpen	
4. Set the search constraint for the row to be updated or deleted	SaCursorEqual, SaCursorAtleast, SaCursorAtmost	
5. Start a search for the row to be updated or deleted	SaCursorSearch	
6. Fetch the row to be updated or deleted	SaCursorNext	
7. Perform actual update or delete	SaCursorUpdate or SaCursorDelete	For updates, the new values need to be in variables bound in step 2.
8. Free the cursor	SaCursorFree	

The following code sample excerpt demonstrates how to update a row in a table using SaCursorUpdate. Note that in the code, the new values for the update are in variables which are bound to the columns of the table using SaCursorColInt and SaCursorColStr after the cursor is created.

```
scur = SaCursorCreate(scon, "SAEXAMPLE");

/* Bind variables to columns INTC and CHARC of test table. */
SaCursorColInt(scur, "INTC", &intc);
SaCursorColStr(scur, "CHARC", &str);

/* Open the cursor. */
SaCursorOpen(scur);

/* Set search constraint. */
str = "D";
SaCursorEqual(scur, "CHARC");

/* Start a search. */
SaCursorSearch(scur);

/* Fetch the column. */
SaCursorNext(scur);
```

```
/* Update the current row in the cursor. */
intc = 1000;
str = "D Updated";
SaCursorUpdate(scur);

/* Close the cursor. */
SaCursorFree(scur);
```

Reading Data by Using solidDB SA without SQL

solidDB SA functions required for query operations are listed in this topic.

With solidDB SA, data is queried using cursors. The query data is found in a way similar to update and delete operations. A cursor is created to a specific table, variables are bound to columns of the table, and the cursor is then opened. The constraints for finding the rows for the query are set before starting the actual search. If more than one row is found, each row must be fetched separately. After all the rows are fetched, the cursor needs to be freed.

Basically, all solidDB SA queries use the solidDB optimizer in a way similar to SQL-based queries. The index selection strategy is the same as in SQL. The only exception is that the solidDB SA search uses ORDER BY for selecting an index. This means that an index that best fits ORDER BY is the one selected. If two indices are equally good, then the one with a smaller cost is selected. The query is optimized each time SaCursorSearch is called.

Note: There is no way to use optimizer hints functionality when Solid SA is used.

Table 47. Query Operation Steps

Steps	SA Function(s)	Comment
1. Create a cursor	SaCursorCreate	
2. Binding variables to cursor	SaCursorColInt, SACursorColStr and others for other data types	
3. Open the cursor	SaCursorOpen	
4. Set the search constraint for the row to be queried	SaCursorEqual, SaCursorAtleast, SaCursorAtmost	
5. Start a search for the row to be queried	SaCursorSearch	
6. Fetch the row(s) that match the given criteria	SaCursorNext	Perform this in a loop if necessary
7. Free the cursor	SaCursorFree	

Example

```
/* Create cursor to a database table. */
scur = SaCursorCreate(scon, "SAEXAMPLE");

/* Bind variables to columns of test table. */
```

```
    rc = SaCursorColInt(scur, "INTC", &intc);
    rc = SaCursorColStr(scur, "CHARC", &str);

    /* Open the cursor. */
    rc = SaCursorOpen(scur);
    assert(rc == SA_RC_SUCC);

    /* Set search constraints. */
    str = "A";
    rc = SaCursorAtleast(scur, "CHARC");
    str = "C";
    rc = SaCursorAtmost(scur, "CHARC");

    /* Set ordering criteria. */
    rc = SaCursorAscending(scur, "CHARC");

    /* Start a search. */
    rc = SaCursorSearch(scur);

    /* Fetch the rows. */
    for (i = 1; i <= 3; i++) {
        rc = SaCursorNext(scur);
        switch (intc) {
            case 1:
                assert(strcmp(str, "A") == 0);
                break;
            case 2:
                assert(strcmp(str, "B") == 0);
                break;
            case 3:
                assert(strcmp(str, "C") == 0);
                break;
        }

     }

    /* Close the cursor. */
    SaCursorFree(scur);
}
```

Running SQL Statements by Using solidDB SA

In addition to bypassing SQL Parser and SQL interpretation, solidDB SA also allows limited execution of SQL statements directly using function SaSQLExecDirect.

This function is designed to execute simple SQL statements, such as CREATE TABLE. If you need to retrieve SQL result sets, you must use another programming interface such as ODBC.

Example

```
/* Create test table and index. */
SaSQLExecDirect(scon,
        "CREATE TABLE SAEXAMPLE(INTC INTEGER, CHARC VARCHAR)");
SaSQLExecDirect(scon,
        "CREATE INDEX SAEXAMPLE_I1 ON SAEXAMPLE (CHARC)");
```

Transactions and autocommit mode

By default, solidDB SA runs in autocommit mode.

Autocommit mode is switched off by calling the function SaTransBegin, which explicitly begins a transaction. In this mode, the transaction is committed using the SaTransCommit function or rolled back using SaTransRollback.

Note: After the transaction is committed, solidDB SA returns to its autocommit mode setting.

In autocommit mode, the transaction is committed immediately after an insert (SaCursorInsert), update (SaCursorUpdate), or delete (SaCursorDelete). Note that even when using SaArrayInsert, each individual record is inserted in a separate transaction if autocommit is used (see “SaArrayInsert” on page 106 for more details). To improve performance when inserting multiple rows with the SaArrayInsert function, put multiple inserts into a single transaction by using SaTransBegin and SaTransCommit.

Handling database errors

This section contains information about database error handling.

solidDB SA does not provide ODBC-like error processing capability. Generally, solidDB SA functions return SA_RC_SUCC or a pointer to the requested object if successful. If not successful, then the return value is either one of the solidDB SA error codes (see the table in the following section) or NULL. If the error is a database error, the error text is returned by the SaErrorInfo function.

```
if (scon == NULL) {
        /* Connect failed, display connect error text. */
        char* errstr;
        SaErrorInfo(NULL, &errstr, NULL);
        printf("%s\n", errstr);
        return(1);
    }
```

The function SaCursorErrorInfo returns error text if the last cursor operation failed. Note that SaErrorInfo has a connection parameter and thus returns the last error applicable to that connection, while SaCursorErrorInfo has a cursor parameter and thus returns the last error of that cursor.

Error Code and Messages for solidDB SA Functions

Following are the possible return codes for solidDB SA functions. All of these error codes are defined in the `sa.h` file.

Table 48. solidDB SA Function Return Codes

Error Code	Meaning
SA_RC_SUCC	Operation was successful
SA_RC_END	Operation has completed
SA_ERR_FAILED	Operation failed
SA_ERR_CURNOTOPENED	Cursor is not open
SA_ERR_CUROPENED	Cursor is open

Table 48. solidDB SA Function Return Codes (continued)

Error Code	Meaning
SA_ERR_CURNOSEARCH	No active search in cursor
SA_ERR_CURSEARCH	There is active search in cursor
SA_ERR_ORDERBYILL	Illegal "order by" specification
SA_ERR_COLNAMEILL	Illegal column name
SA_ERR_CONSTRILL	Illegal constraint
SA_ERR_TYPECONVILL	Illegal type conversion
SA_ERR_UNIQUE	Unique constraint violation
SA_ERR_LOSTUPDATE	Concurrency conflict, two transactions updated or deleted the same row
SA_ERR_SORTFAILED	Failed to sort the search result set
SA_ERR_CHSETUNSUPP	Unsupported character set
SA_ERR_CURNOROW	No current row in cursor
SA_ERR_COLISNOTNULL	NULL value given for a NOT NULL column
SA_ERR_LOCALSORT	Result set is sorted locally, cannot update or delete the row
SA_ERR_COMERROR	Communication error, connection is lost
SA_ERR_NOSTRCONSTR	String for constraint is missing.
SA_ERR_ILLENUMVAL	Illegal numeric value
SA_ERR_COLNOTBOUND	Column is not bound
SA_ERR_CALLNOSUP	Operation is not supported*
SA_ERR_RPCPARAM	RPC parameter error
SA_ERR_TABLENOTFOUND	Table not found
SA_ERR_READONLY	Connection is read only
SA_ERR_ILLPARAMCOUNT	Wrong number of parameters
SA_ERR_INVARG	Invalid argument
SA_ERR_INVCALLSEQ	Invalid call sequence

Note: SaArray* functions are not supported in linked library access; they work only with the network client library. They return SA_ERR_CALLNOSUP with linked library access.

Special notes about solidDB SA

This topic contains important information and restrictions about solidDB SA.

solidDB SA and Binary Large Objects (BLOBs)

Currently, solidDB SA does not support BLOB streams and the maximum size of an attribute value is limited to 32K.

SaCursorCol* Functions and solidDB SQL Supported Datatypes

The SaCursorColXXX() functions bind a variable of type XXX to a specified column. For example, the SaCursorColInt function binds a variable of type int to a specified column. When you bind a variable to a column, the variable and column usually have corresponding types; for example, you usually bind an int C variable to an INT SQL column. However, it is not absolutely required that the data type of the column and the data type of the bound variable be equivalent. For example, you could bind a C int variable to an SQL FLOAT, but you would risk losing precision (or even overflowing or underflowing) as data was transferred back and forth.

The SaCursorCol* functions support the SQL datatypes listed in the following table.

Table 49. Supported SQL Datatype

SaCursolCol* Function	**TINYINT**	**SMALLINT**	**INTEGER**	**REAL**	**FLOAT**	**DOUBLE**	**DECIMAL**	**NUMERIC**	**CHAR**	**VARCHAR**	**LONGVARCHAR**	**WCHAR**	**WVARCHAR**	**LONGWVARCHAR**	**BINARY**	**VARBINARY**	**LONGVARBINARY**	**DATE**	**TIME**	**TIMESTAMP**
SaCursorColInt	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
SaCursor ColLong	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
SaCursor ColFloat	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
SaCursor ColDouble	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
SaCursorColStr									X	X	X									
SaCursorCol Date									X	X	X	X	X	X				X		X
SaCursor ColTime									X	X	X	X	X	X					X	X

Table 49. Supported SQL Datatype (continued)

SaCursolCol* Function	**TINYINT**	**SMALLINT**	**INTEGER**	**REAL**	**FLOAT**	**DOUBLE**	**DECIMAL**	**NUMERIC**	**CHAR**	**VARCHAR**	**LONGVARCHAR**	**WCHAR**	**WVARCHAR**	**LONGWVARCHAR**	**BINARY**	**VARBINARY**	**LONGVARBINARY**	**DATE**	**TIME**	**TIMESTAMP**
SaCursor ColTimestamp									X	X	X	X	X	X					X	X
SaCursor ColData															X	X	X			
SaCursor ColDynData		X	X	X	X	X	X	X	X	X	X				X	X	X			
SaCursor ColFixStr		X	X	X	X	X	X	X	X	X	X				X	X	X	X	X	X
SaCursor ColDynStr		X	X	X	X	X	X	X	X	X	X	X	X							

Note: Keep in mind that, as in other APIs, the success of some conversions in solidDB SA depend on declared values. For example, SaCursorCollInt is only able to handle the SQL datatype CHAR (as in 'foo') if the actual value of the field is an integer (as in '123').

solidDB SA Function Reference

This topic contains the list of the solidDB SA functions in alphabetic order.

Each description includes the purpose, synopsis, parameters, return value, and comments.

Function Synopsis

The declaration synopsis for the function is:

```
SA_EXPORT_H function(modifier parameter [,...]);
```

where modifier can be:

```
SaConnectT*
SaColSearchT*
SaCursorT*
SaDataTypeT*
SaDateT*
SaDfloatT*
SaDynDataT*
SaDynStrT*
SaChSetT
char*
char**
double*
long*
float*
```

```
int
int*
unsigned*
void
```

Parameters are in italics and are described below.

Parameter Description

In each function description, parameters are described in a table format. Included in the table is the general usage type of the parameter (described in the next section), as well as the use of the parameter variable in the specific function.

Parameter Usage Type

The table below shows the possible usage type for solidDB SA parameters. Note that if a parameter is used as a pointer, it contains a second category of usage to specify the ownership of the parameter variable after the call.

Table 50. solidDB SA Parameter Usage Types

Usage Type	Meaning
in	Indicates the parameter is input.
output	Indicates the parameter is output.
in out	Indicates the parameter is input/output.
take	Applies only to a pointer parameter. It means that the parameter value is taken by the function. The caller cannot reference to the parameter after the function call. The function or an object created in the function is responsible for releasing the parameter when it is no longer needed.
hold	Applies only to a pointer parameter. It means that the function holds the parameter value even after the function call. The caller can reference to the parameter value after the function call and is responsible for releasing the parameter. Typically, this kind of parameter is passed to the constructor of some object which holds the pointer value inside a local data structure. The caller cannot release the parameter until the object that holds the parameter is deleted.
use	Applies only to a pointer parameter. It means that the parameter is just used during the function call. The caller can do whatever it wants with the parameter after the function call. This is the most common type of parameter passing.
ref	Applies only to out parameters. See "Return Value" below for details.
give	Applies only to out parameters. See "Return Value" below for details.

Return value

Each function description indicates if the function returns a value and the type of value that is returned. Return Values can be one of the following values:

- Boolean (TRUE, FALSE),
- int (such as 1, 0),
- SaRetT (error return codes) such as SA_RC_SUCC. For a list of valid error codes, refer to"Handling database errors" on page 100.
- Pointer (out parameter)

The possible return usage types for pointers are:

Table 51. Return Usage Types for Pointers

Usage Type	Meaning
ref	Indicates the caller can only reference the returned value, but cannot release it. Ensure that the returned value is not used after it is released by the object that returned it.
give	Indicates the function gives the returned value to the caller. The caller is responsible for releasing the returned value.

SaArrayFlush

SaArrayFlush flushes the array operation buffer (that is, it sends the data to the server) after a series of calls to SaArrayInsert fills that buffer.

By default, all SA operations, even SaArrayFlush operations, are done in autocommit mode. In autocommit mode, theSaArrayFlush function does not automatically insert all of the array's records in a single transaction; instead, when SaArrayFlush is called, each record's insertion is treated as a separate transaction. To maximize performance, you may want to do an explicit SaTransBegin before you call SaArrayFlush, and do an explicit SaTransCommit after you call SaArrayFlush

SaArray* functions are not supported in linked library access; they work only with the network client library. They return SA_ERR_CALLNOSUP with linked library access.

Synopsis

```
SaRetT SA_EXPORT_H SaArrayFlush(SaConnectT* scon, SaRetT* rctab)
```

The SaArrayFlush function accepts the following parameters:

Table 52. SaArrayFlush Parameters

Parameters	Usage Type	Description
scon	use	Pointer to a connection object

Table 52. SaArrayFlush Parameters (continued)

Parameters	Usage Type	Description
rctab	use	Array of return codes for each array operation If this parameter is non-NULL, the return code of each array operation is returned in rctab[i], where*i* is the order number of the array operation since the last SaArrayFlush.

Return Value

SA_RC_SUCC or error code of first failed array operation.

See also

"SaArrayInsert."

SaArrayInsert

SaArrayInsert inserts an array of values on one network message. This function places the inserted value in the array insert buffer. You can flush the buffer (that is, send the data to the server) using function SaArrayFlush.

SaArrayInsert may also perform an implicit flush if the internal cache becomes full. However, to ensure that all rows are sent to the server, you should call SaArrayFlush after you insert the last record using SaArrayInsert.

Note:

1. By default, all SA operations, even SaArrayInsert and SaArrayFlush operations, are done in autocommit mode. See "SaArrayFlush" on page 105 for an important note about performance.
2. SaArray* functions are not supported in linked library access; they work only with the network client library. They return SA_ERR_CALLNOSUP with linked library access.

Synopsis

```
SaRetT SA_EXPORT_H SaArrayInsert(SaCursorT* scur)
```

The SaArrayInsert function accepts the following parameters:

Table 53. SaArrayInsert Parameters

Parameters	Usage Type	Description
scur	use	Pointer to a cursor object

Return Value

SA_RC_SUCC or error code

See Also

"SaArrayFlush" on page 105.

SaColSearchCreate

SaColSearchCreate starts a column information search for a specified table.

Synopsis

```
SaColSearchT* SA_EXPORT_H SaColSearchCreate(
    SaConnectT* scon,
    char* tablename)
```

The SaColSearchCreate function accepts the following parameters:

Table 54. SaColSearchCreate Parameters

Parameters	Usage Type	Description
scon	In	Pointer to a connection object
tablename	In	Table name

Return Value

Pointer to the column search object, or NULL if table does not exist.

SaColSearchFree

SaColSearchFree releases the column search object.

Synopsis

```
void SA_EXPORT_H SaColSearchFree(SaColSearchT* colsearch)
```

The SaColSearchCreate function accepts the following parameters:

Table 55. SaColSearchCreate Parameters

Parameters	Usage Type	Description
colsearch	In, take	Column search pointer

Return Value

None

SaColSearchNext

SaColSearchNext returns information about the next column in the table.

Synopsis

```
int SA_EXPORT_H SaColSearchNext(
    SaColSearchT* colsearch,
    char** p_colname,
    SaDataTypeT* p_coltype)
```

The SaColSearchNext function accepts the following parameters:

Table 56. SaColSearchNext Parameters

Parameters	Usage Type	Description
colsearch	in, use	Column search pointer
p_colname	out, ref	Pointer to the local copy of the column name is stored into * p_colname
p_coltype	out	Type of column is stored into * p_coltype. See the `sa.h` file for a description of the SaDataTypeT data type and the valid values that it can hold.

Return Value

Table 57. SaColSearchNext Return Value

Value	Description
1	Next column found, parameters updated.
0	0 No more columns, parameters not updated. The function also will return 0 if the input parameters are invalid.

SaConnect

SaConnect creates a connection to the solidDB server. Several connections can be active at the same time, but operations in different connections are executed in separate transactions.

Synopsis

```
SaConnectT* SA_EXPORT_H SaConnect(
    char* servername,
    char* username,
    char* password)
```

The SaConnect function accepts the following parameters:

Table 58. SaConnect Parameters

Parameters	Usage Type	Description
servername	in, use	Server name. An empty servername connects to the linked server.
username	in, use	User name
password	in, use	Password

Return Value

Table 59. SaConnect Return Value

Return Usage Type	Description
give	Connect pointer, or if connection failed, NULL.

SaCursorAscending

SaCursorAscending specifies ascending order criteria for a column.

To sort by more than one column, you must call this function once for each column. If there is no key (primary key or index) on the column, then the rows are sorted locally (on the client) rather than on the server side.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorAscending(
   SaCursorT* scur,
   char* colname)
```

The SaCursorAscending function accepts the following parameters:

Table 60. SaCursorAscending parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name

Return Value

SA_RC_SUCC or error code

SaCursorAtleast

SaCursorAtleast specifies the Atleast criterion for a column. Atleast criterion means that the column value must be greater than or equal to the Atleast value. The Atleast value is taken from the user variable currently bound to the column.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorAtleast(
    SaCursorT* scur,
    char* colname)
```

The SaCursorAtleast function accepts the following parameters:

Table 61. SaCursorAtleast Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name

Return Value

SA_RC_SUCC or error code

SaCursorAtmost

SaCursorAtmost specifies the Atmost criterion for a column. Atmost criterion means that the column value must be less than or equal to the Atmost value. The Atmost value is taken from the user variable currently bound to the column.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorAtmost(
    SaCursorT* scur,
    char* colname)
```

The SaCursorAtmost function accepts the following parameters:

Table 62. SaCursorAtmost Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name

Return Value

SA_RC_SUCC or error code

SaCursorBegin

SaCursorBegin positions the cursor to the beginning of the set. The subsequent call to the SaCursorNext function returns the first row.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorBegin(
    SaCursorT* scur)
```

The SaCursorBegin function accepts the following parameters:

Table 63. SaCursorBegin Parameters

Parameters	Usage Type	Description
scur	use	Pointer to a cursor object

Return Value

SA_RC_SUCC or error code

SaCursorClearConstr

SaCursorClearConstr clears all search constraints from a cursor.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorClearConstr(
    SaCursorT* scur)
```

The SaCursorClearConstr function accepts the following parameters:

Table 64. SaCursorClearConstr Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Return Value

SA_RC_SUCC or error code

SaCursorColData

SaCursorColData binds a user variable to a database column.

The bound variable may be used either as an "input" parameter or an "output" parameter. An "input" parameter passes data values from the client to the server for operations such as inserts and updates, and for search constraints. An "output" parameter holds values read by the server during searches. For example, to INSERT data, the user first binds the variables, then stores values into the bound variables before the actual INSERT; those values are then copied into the database when the INSERT is performed. Similarly, during a fetch operation, when the next row is retrieved, the values in the columns of that row are copied into bound variables so that the client program can see them.

A variable may be used multiple times after a single binding. For example, if you wanted to insert multiple rows, you might create a loop in which you store appropriate values in the bound variable and then invoke the INSERT operation. The "bind" operation would only need to be done once before the loop; it would not need to be executed inside the loop for each INSERT operation. Similarly, after binding the variables once, you could retrieve many rows (one at a time) using the SaCursorNext function. Each time that you retrieved a row, its values would be copied into the bound variables. Note that the address of the data buffer does not change; only the value stored there changes each time what you call SaCursorNext.

If the column has been set as a search constraint (rather like using a WHERE clause in a SELECT statement), then the value for this constraint is set to the value pointed to by the user data variable whose address is passed as dataptr. For example, if the function SaCursorEquals has been called for the column, then the server retrieves only the rows whose value exactly matches the current value of the bound variable. Note that the search constraints are set up for the search operations (SaCursorSearch, followed by calls to SaCursorNext) but may actually be used to set the cursor to the correct position for other operations (such as SaCursorUpdate or SaCursorDelete).Typically, updates are combined with searches to update only some of the rows. This means that the values for columns which have search constraints are used to define the affected rows (in effect the "WHERE" clause in SQL) and other bound variables are used to define the new values for the rest of the columns. Note that the same bound variable can be used in both the search constraint and in the update/insert operation (just as the same column may be used in both the WHERE clause and the "UPDATE ... SET col = value" clause of an SQL UPDATE statement). If the same bound variable is used in both the search constraint and to convey data back and forth between the client and the server, the search constraint does not change each time that the data in the bound variable is updated; the server uses the value that was in the bound variable at the time that the search constraint was created (e.g. when functions like SaCursortAtmost() were called).

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. In insert and update operations the new value for the column is taken from the user variable.

When the bound variable is used as an "in" parameter (for example, in INSERT or UPDATE operations), the user is responsible for the allocation and freeing of the buffer. When a bound variable is used as an "out" parameter, the SA layer allocates and frees the buffers. When the variable is used as an "out" parameter, the value stored to the user variable is a pointer to a buffer that contains a local copy of the column data. After each row is retrieved, that row's value will be copied to this buffer. The pointer to this buffer is valid until the next SaCursorOpen or SaCursorFree call, after which the pointer should not be referenced.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColData(
    SaCursorT* scur,
    char* colname,
    char** dataptr,
    unsigned* lenptr)
```

The SaCursorColData function accepts the following parameters:

Table 65. SaCursorColData Parameters

Parameters	**Usage Type**	**Description**
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
dataptr	in, hold	Pointer to the user variable
lenptr	in, hold	Pointer to variable used to hold length of data

Return Value

SA_RC_SUCC or error code.

SaCursorColDate

SaCursorColDate binds a user variable of type SaDateT to a database column.

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. In insert and update operations the new value for the column is taken from the user variable.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColDate(
SaCursorT* scur,
char* colname,
SaDateT* dateptr)
```

The SaCursorColDate function accepts the following parameters:

Table 66. SaCursorColDate Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
dateptr	in, hold	Pointer to the user variable

Return Value

SA_RC_SUCC or error code.

See Also

See "SaCursorColData" on page 111 for a more detailed discussion of binding variables.

SaCursorColDateFormat

SaCursorColDateFormat binds date format string to a database column.

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. Depending on the column data type, the format string should be date, time, or timestamp format.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColDateFormat(
    SaCursorT* scur,
    char* colname,
    char* dtformat)
```

The SaCursorColDateFormat function accepts the following parameters:

Table 67. SaCursorColDateFormat parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
dtformat	in, hold	Date/time/timestamp format string

Return Value

SA_RC_SUCC or error code.

See Also

See "SaCursorColDynData" on page 115 for a more detailed discussion of binding variables. For explanation of possible date/time/timestamp formats, see "SaDateSetAsciiz" on page 135.

SaCursorColDfloat

SaCursorColDfloat binds a user variable of type SaDfloatT to a database column.

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. In insert and update operations, the new value for the column is taken from the user variable.

Note: SaDFloatT corresponds to the SQL data type DECIMAL (not FLOAT).

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColDfloat(
    SaCursorT* scur,
    char* colname,
    SaDfloatT* dfloatptr)
```

The SaCursorColDfloat function accepts the following parameters:

Table 68. SaCursorColDfloat Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
dtfloatptr	in, hold	Pointer to the user variable

Return Value

SA_RC_SUCC or error code.

See Also

"SaCursorColDouble."

"SaCursorColFloat" on page 117.

See "SaCursorColData" on page 111 for a more detailed discussion of binding variables.

SaCursorColDouble

SaCursorColDouble binds a user variable of type double to a database column.

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. In insert and update operations the new value for the column is taken from the user variable.

Note: The C-language data type "double" is equivalent to SQL data type "FLOAT".

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColDouble(
    SaCursorT* scur,
    char* colname,
    double* doubleptr)
```

The SaCursorColDouble function accepts the following parameters:

Table 69. SaCursorColDouble Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
doubleptr	in, hold	Pointer to the user variable

Return Value

SA_RC_SUCC or error code.

See Also

“SaCursorColFloat” on page 117.

“SaCursorColDfloat” on page 114.

See “SaCursorColDynData” for a more detailed discussion of binding variables.

SaCursorColDynData

SaCursorColDynData binds a user variable of type SaDynDataT to a database column.

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. In insert and update operations, the new value for the column is taken from the user variable.

In search operations, the column data is stored to the SaDynDataT variable using function SaDynDataMove, which overwrites the old data. The user is responsible for releasing the SaDynDataT variable after the search ends using function SaDynDataFree.

Dynamic data objects (SaDynDataT) are an abstraction that simplifies the handling of variable length data. Although dynamic data can be used with all types of data, it is best fit for variable length data (VARBINARY, LONG VARBINARY, VARCHAR, LONG VARCHAR, etc.).

The memory management of the data object is hidden inside the object. Dynamic data objects have two externally-visible attributes: the data and the length. Typically, the functions SaDynDataMove and SaDynDataAppend are used to set and modify the data value inside the dynamic data object. More memory will be automatically allocated when necessary and all the associated memory will be automatically deallocated when the dynamic data object is disposed of using SaDynDataFree. The user can access the data or the length using the respective functions SaDynDataGetData and SaDynDataGetLen.

The use of SaDynDataMove and SaDynDataAppend may not be feasible when the data already exists completely in a memory buffer. In addition to increasing the memory usage by keeping two copies of the same data, the overhead of the memory copy may be significant if the buffers are large. Therefore, it may be wise to directly assign the data pointer by using SaDynDataMoveRef (rather than copying by using SaDynDataMove). In this case, the user may modify or deallocate the memory buffer only after the dynamic data object itself has been freed.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColDynData(
    SaCursorT* scur,
    char* colname,
    SaDynDataT* dd)
```

The SaCursorColDynData function accepts the following parameters:

Table 70. SaCursorColDynData Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
dd	in, hold	Pointer to the user variable

Return Value

SA_RC_SUCC or error code.

SaCursorColDynStr

SaCursorColDynStr binds a user variable of type SaDynStrT to a database column.

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. In insert and update operations, the new value for the column is taken from the user variable.

In search operations, the column data is stored to the SaDynStrT variable using function SaDynStrMove, which overwrites the old data. The user is responsible for releasing the SaDynStrT variable after the search ends using function SaDynStrFree.

The user may bind an SaDynStrT variable to any type of column (not just character columns) and the data will be converted back and forth between the column type and the Dynamic String type.

Dynamic String objects (SaDynStrT) are an abstraction that simplifies the handling of variable length strings. Typically, the functions SaDynStrMove and SaDynStrAppend are used to set and modify the data value inside the dynamic string object. More memory will be automatically allocated when necessary and all the associated memory will be automatically deallocated when the dynamic data object is disposed of using SaDynStrFree.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColDynStr(
    SaCursorT* scur,
    char* colname,
    SaDynStrT* ds)
```

The SaCursorColDynStr function accepts the following parameters:

Table 71. SaCursorColDynStr Parameters

Parameters	**Usage Type**	**Description**
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
ds	in, hold	Pointer to the user variable

Return Value

SA_RC_SUCC or error code.

See Also

See “SaCursorColDynData” on page 115 for a more detailed discussion of binding variables.

SaCursorColFloat

SaCursorColFloat binds a user variable of type float to a database column.

After the variable has been bound, it can be used to hold a value that will be written to or read from a column, or that will be used to constrain a search operation (e.g. as part of the equivalent of a WHERE clause in SQL). In search operations, the user variable is updated to contain the value read from the current row that has been retrieved. Also, if search criteria are involved, this function can be used to pass the values for them. In update and insert operations, the new value is taken from the bound user variable and then written to the column in the database.

Note: The C-language "float" data type corresponds to the SQL "SMALLFLOAT" data type, not the SQL "FLOAT" data type.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColFloat(
    SaCursorT* scur,
    char* colname,
    float* floatptr)
```

The SaCursorColFloat function accepts the following parameters:

Table 72. SaCursorColFloat Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
floatptr	in, hold	Pointer to the user variable

Return Value

SA_RC_SUCC or error code.

See Also

"SaCursorColDouble" on page 114.

"SaCursorColDfloat" on page 114.

See "SaCursorColDynData" on page 115 for a more detailed discussion of binding variables.

SaCursorColInt

SaCursorColInt binds a user variable of type int to a database column.

After the variable has been bound, it can be used to hold a value that will be written to or read from a column, or that will be used to constrain a search operation (e.g. as part of the equivalent of a WHERE clause in SQL). In search operations, the user variable is updated to contain the value read from the current row that has been retrieved. Also, if search criteria are involved, this function can be used to pass the values for them. In update and insert operations, the new value is taken from the bound user variable and then written to the column in the database.

Note: The C-language "int" data type is platform-dependent, while the SQL data types (TINYINT, SMALLINT, INT, and BIGINT) are platform-independent. You must be careful to map the appropriate C-language data type and value to the corresponding SQL data type.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColInt(
SaCursorT* scur,
char* colname,
int* intptr)
```

The SaCursorColInt function accepts the following parameters:

Table 73. SaCursorColInt Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Table 73. SaCursorColInt Parameters (continued)

Parameters	Usage Type	Description
colname	in, use	Column name
intptr	in, hold	Pointer to the user variable

Return Value

SA_RC_SUCC or error code.

See Also

See “SaCursorColDynData” on page 115 for a more detailed discussion of binding variables.

SaCursorColLong

SaCursorColLong binds a user variable to a database column.

After the variable has been bound, it can be used to hold a value that will be written to or read from a column, or that will be used to constrain a search operation (e.g. as part of the equivalent of a WHERE clause in SQL). In search operations, the user variable is updated to contain the value read from the current row that has been retrieved. Also, if search criteria are involved, this function can be used to pass the values for them. In update and insert operations, the new value is taken from the bound user variable and then written to the column in the database.

Note: The C-language "long" data type is platform-dependent, while the SQL data types (TINYINT, SMALLINT, INT, and BIGINT) are platform-independent. You must be careful to map the appropriate C-language data type and value to the corresponding SQL data type.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColLong(
    SaCursorT* scur,
    char* colname,
    long* longptr)
```

The SaCursorColLong function accepts the following parameters:

Table 74. SaCursorColLong Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
longptr	in, hold	Pointer to the user variable

Return Value

SA_RC_SUCC or error code.

See Also

See “SaCursorColDynData” on page 115 for a more detailed discussion of binding variables.

SaCursorColNullFlag

SaCursorColNullFlag binds a NULL value flag to a column.

If the column value is NULL, then * p_isnullflag has a value 1,otherwise the value is 0. The * p_isnullflag value is updated automatically during fetch operations. In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. During insert and update, a NULL value is inserted to the database if *p _isnullflag is not zero.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColNullFlag(
SaCursorT* scur,
char* colname,
int* p_isnullflag)
```

The SaCursorColNullFlag function accepts the following parameters:

Table 75. SaCursorColNullFlag Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
p_isnullflag	in, hold	Pointer to an integer variable into where the NULL status is stored during fetch operations, and from where the NULL status is taken during insert and update operations.

Return Value

SA_RC_SUCC or error code.

See Also

See “SaCursorColDynData” on page 115 for a more detailed discussion of binding variables.

SaCursorColStr

SaCursorColStr binds a user variable to a database column.

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. In insert and update operations the new value for the column is taken from the user variable.

In search operations, the value stored to the user variable is a pointer to a local copy of the column data. The data pointer is valid until the next SaCursorOpen or SaCursorFree call, after which the pointer should not be referenced.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColStr(
SaCursorT* scur,
char* colname,
char** strptr)
```

The SaCursorColStr function accepts the following parameters:

Table 76. SaCursorColStr Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
strptr	in, hold	Pointer to the user variable.

Return Value

SA_RC_SUCC or error code.

See Also

See "SaCursorColDynData" on page 115 for a more detailed discussion of binding variables.

SaCursorColTime

SaCursorColTime binds a user variable of type SaDateT to a database column.

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. In insert and update operations, the new value for the column is taken from the user variable.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColTime(
    SaCursorT* scur,
    char* colname,
    SaDateT* timeptr)
```

Note: The data type of timeptr is indeed SaDateT; there is no separate SaTimeT for time data.

The SaCursorColTime function accepts the following parameters:

Table 77. SaCursorColTime parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Table 77. SaCursorColTime parameters (continued)

Parameters	Usage Type	Description
colname	in, use	Column name
timeptr	in, hold	Pointer to the user variable.

Return Value

SA_RC_SUCC or error code.

See Also

See "SaCursorColDynData" on page 115 for a more detailed discussion of binding variables.

SaCursorColTimestamp

SaCursorColTimestamp binds a user variable of type SaDateT to a database column.

In search operations, the user variable is updated to contain the value of the current row. Also, if search criteria are involved, this function is used to pass the values for them. In insert and update operations the new value for the column is taken from the user variable.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorColTimestamp(
    SaCursorT* scur,
    char* colname,
    SaDateT* timestampptr)
```

Note: The data type of timeptr is indeed SaDateT; there is no separate SaTimestampT for timestamp data.

The SaCursorColTimestamp function accepts the following parameters:

Table 78. SaCursorColTimestamp parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
timestampptr	in, hold	Pointer to the user variable.

Return Value

SA_RC_SUCC or error code.

See Also

See “SaCursorColDynData” on page 115 for a more detailed discussion of binding variables.

SaCursorCreate

SaCursorCreate creates a cursor to a table specified by table name. The operation fails if the table does not exist.

Synopsis

```
SaCursorT* SA_EXPORT_H SaCursorCreate(
    SaConnectT* scon,
    char* tablename)
```

The SaCursorCreate function accepts the following parameters:

Table 79. SaCursorCreate Parameters

Parameters	Usage Type	Description
scon	in, hold	Pointer to a connection object
tablename	in, use	Table name

Return Value

The parameter *scon* has the Usage Type "hold" because the created cursor object keeps referencing the scon object even after the function call has returned.

Table 80. Return Value

Return Usage Type	Description
give	Pointer to the cursor object, or NULL if table does not exist.

SaCursorDelete

SaCursorDelete deletes the current row in a cursor from the database. The cursor must be positioned to a row.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorDelete(SaCursorT* scur)
```

The SaCursorDelete function accepts the following parameters:

Table 81. SaCursorDelete parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Return value

SA_RC_SUCC or error code.

SaCursorDescending

SaCursorDescending specifies descending sorting criterion for a column.

To sort by more than one column, you must call this function once for each column.

If there is no key (primary key, or index) on the column, then the rows are sorted locally (on the client) rather than on the server side.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorDescending(
    SaCursorT* scur,
    char* colname)
```

The SaCursorDescending function accepts the following parameters:

Table 82. SaCursorDescending parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name

Return value

SA_RC_SUCC or error code.

SaCursorEnd

SaCursorEnd positions the cursor to the end of the set. A subsequent call to SaCursorPrev will position the cursor to the last row in the set.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorEnd(
    SaCursorT* scur)
```

The SaCursorEnd function accepts the following parameters:

Table 83. SaCursorEnd parameters

Parameters	Usage Type	Description
scur	use	Pointer to a cursor object

Return value

SA_RC_SUCC or error code.

SaCursorEqual

SaCursorEqual specifies an equal search criterion for a column.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorEqual(
    SaCursorT* scur,
    char* colname)
```

The SaCursorEqual function accepts the following parameters:

Table 84. SaCursorEqual parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name

Return value

SA_RC_SUCC or error code.

SaCursorErrorInfo

SaCursorErrorInfo returns error information from the last operation in the cursor.

Synopsis

```
bool SA_EXPORT_H SaCursorErrorInfo(
SaCursorT* scur,
char** errstr,
int* errcode)
```

The SaCursorErrorInfo function accepts the following parameters:

Table 85. SaCursorErrorInfo parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
errstr	out, ref	If non-NULL, pointer to a local copy of an error string is stored into * errstr.
errcode	out	If non-NULL, error code is stored into * errcode.

Return value

TRUE If there are errors, errstr and errcode are updated.

FALSE If there are no errors, errstr and errcode are not updated.

SaCursorFree

SaCursorFree releases a cursor. After this call the cursor pointer is invalid.

Synopsis

```
void SA_EXPORT_H SaCursorFree(SaCursorT* scur)
```

The SaCursorFree function accepts the following parameters:

Table 86. SaCursorFree parameters

Parameters	Usage Type	Description
scur	in, take	Pointer to a cursor object

Return value

None.

SaCursorInsert

SaCursorInsert inserts a new row into the database. Column values for the new row are taken from the user variables bound to columns. The cursor must be opened before new rows can be inserted.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorInsert(SaCursorT* scur)
```

The SaCursorInsert function accepts the following parameters:

Table 87. SaCursorInsert parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Return value

SA_RC_SUCC or error code.

SaCursorLike

SaCursorLike specifies a like criterion for a column.

The value cannot contain any wild card characters like '_' or '%' in SQL. If such characters exist in the column value, they are quoted with escape characters by the system. Thus, the like value is effectively the same as the SQL like with no wild card characters ending with a '%' character. For example, if you specify that the engine should search for "MARK" in the column, then the engine will find all values that start with "MARK", such as "MARK", "MARK SMITH", and "MARKETING".

The like value is taken from the user variable bound to the column.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorLike(
    SaCursorT* scur,
    char* colname,
    int likelen)
```

The SaCursorLike function accepts the following parameters:

Table 88. SaCursorLike parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name
likelen	in	Length of like part (excluding the string terminator)

Return value

SA_RC_SUCC or error code.

SaCursorNext

SaCursorNext fetches the next row from the database. All user variables bound to columns are updated.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorNext(SaCursorT* scur)
```

The SaCursorNext function accepts the following parameters:

Table 89. SaCursorNext parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Return value

SA_RC_SUCC Next row found

SA_RC_END End of search

SaCursorOpen

SaCursorOpen opens a cursor.

All SaCursorColXXX operations must be done before the cursor is opened. When the cursor is opened, possible existing search is terminated. Also, all search criteria specified for the cursor are cleared.

After the cursor is opened, user can insert new rows to the cursor or specify search criteria. Cursor must be opened before a search can be started.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorOpen(SaCursorT* scur)
```

The SaCursorOpen function accepts the following parameters:

Table 90. SaCursorOpen parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Return value

SA_RC_SUCC or error code.

SaCursorOrderbyVector

SaCursorOrderbyVector is used to specify the order of columns used in a search.

The initial values are used as a vector of values to specify the starting position for the search in the key. The initial value is used only for the starting point selection in the key; after that, the initial values are not checked against the column values. If several criteria are given, they are solved in the given order. A proper key must exist for the ordering.

The initial value is taken from the user variable bound to the column.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorOrderbyVector(
    SaCursorT* scur,
    char* colname)
```

The SaCursorOrderbyVector function accepts the following parameters:

Table 91. SaCursorOrderbyVector parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
colname	in, use	Column name

Return Value

SA_RC_SUCC or error code.

SaCursorOrderbyVector example

```
/* These variables will be bound to the columns named "I" and "J" */
int i, j;
/* Bind variables to columns in this cursor. */
SaCursorColStr(scur, "I", 'i);
SaCursorColStr(scur, "J", 'j);
/* Set the values that we want to use in the search. */
i = 2;
j = 1;
/* Specify the order of the columns. */
SaCursorOrderByVector(scur, "I");
SaCursorOrderByVector(scur, "J");
/* Search the cursor for matching values. */
SaCursorSearch(scur);
```

The preceding would be the equivalent of the following SQL WHERE clause:

```
...WHERE (i,j) >= (2,1)
```

SaCursorPrev

SaCursorPrev fetches the previous row from the database. All user variables currently bound to columns are updated.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorPrev(SaCursorT* scur)
```

The SaCursorPrev function accepts the following parameters:

Table 92. SaCursorPrev parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Return value

SA_RC_SUCC Previous row found

SA_RC_END Beginning of search (we are already at the first row, so there is no previous row).

Note: SA_RC_END can apply to either end (start or finish) of the cursor.

SaCursorReSearch

SaCursorReSearch starts a new search using old search criteria.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorReSearch(SaCursorT* scur)
The SaCursorReSearch function accepts the following
parameters:
```

Table 93. SaCursorReSearch Parameters

Parameters	Usage Type	Description
scur	use	Pointer to a cursor object

Return value

SA_RC_SUCC, SA_RC_END, or error code. See "Handling database errors" on page 100 for a list of error codes.

SaCursorSearch

SaCursorSearch starts a search in a cursor. After the search is started, the user can fetch rows from the database. Every search is executed as a separate transaction and it does not see any changes made by the current user or any other user after the search is started.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorSearch(SaCursorT* scur)
```

The SaCursorSearch function accepts the following parameters:

Table 94. SaCursorSearch parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Return value

SA_RC_SUCC, SA_RC_END or error code

SaCursorSearchByRowid

SaCursorSearchByRowid starts a new search where the row specified by rowid belongs to the search set.

SaCursorSearchByRowid searches only according to rowid, so it returns one row or zero rows. Previous search constraints are not removed, and they become effective in the next SaCursorReSearch call.

To get the rowid for a particular record, read the value of the rowid column. Every table has a rowid column; you do not need to explicitly create a rowid column within a CREATE TABLE or ALTER TABLE statement.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorSearchByRowid(
    SaCursorT* scur,
    void* rowid,
    int rowidlen)
```

The SaCursorSearchByRowid function accepts the following parameters:

Table 95. SaCursorSearchByRowid parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
rowid	in, use	Pointer to a data area containing rowid. The rowid should be in the form of a string (char *) despite the fact that it is declared as "void *".
rowidlen	in	Length of data (string) pointed to by the rowid parameter

Return Value

SA_RC_SUCC, SA_RC_END, or error code.

SaCursorSearchReset

SaCursorSearchReset resets a search cursor.

The old search constraints are used, but their values are read again from the user buffers (i.e. the parameters). This allows you to increase performance in situations where you want to repeat a search using a query that is identical except for the specific values used.

For example, suppose that a particular user or connection always searches a particular table based on the ID column in that table, but uses a different ID value in each search. Instead of creating a "new" query to search for the next ID, you can reset the existing query and use a new value.

As an example, suppose that your existing code looks similar to the following:

```
...
/* Bind variable(s) to column(s). */
SaCursorColInt(scur, "MY_COL_NAME", &search_parameter1);

/* Repeat a query using different values each time. */
while (there_are_more_values_to_look_for)  {
    /* Set the parameter to the value that you want to search for. */
    search_parameter1 = some_value;
    /* Specify the search criterion. */
    rc = SaCursorEqual(scur, "MY_COL_NAME");
    /* Create new query that uses that search criterion and param value*/
    rc = SaCursorSearch(scur);
    /* Get the row (or rows) that match the search criteria. */
    rc = SaCursorNext(scur);
    /* Process the retrieved data... */
    foo();
    ...
    /* Get rid of the old query before the next loop iteration. */
    rc = SaCursorClearConstr(scur);
    }
...
```

You can improve performance in most cases by changing your code to look like the following example:

```
...
/* Bind variable(s) to column(s). */
SaCursorColInt(scur, "MY_COL_NAME", &search_parameter1);

/* Create a new query. */
rc = SaCursorEqual(scur, "MY_COL_NAME");
rc = SaCursorSearch(scur);
/* Set the parameter to the value that you want to search for. */
search_parameter1 = some_value;

/* Repeat a query using different values each time. */
while (there_are_more_values_to_look_for)  {
    /* Get the row (or rows) that match the search criteria. */
    rc = SaCursorNext(scur);
    /* Process the retrieved data... */
    foo();
    ...
    /* Set the param to the next value that you want to search for. */
    search_parameter1 = some_value;
    /* Reset the existing query to use the latest value in the param. */
    rc = SaCursorSearchReset(scur);
    }
...
```

When you use SaCursorSearchReset(), you no longer have to re-specify the constraint condition ("Equal", in the example above) and call SaCursorSearch() each time.

SaCursorSearchReset resets the cursor to the beginning of the new result set. For example, if you reset a search with no constraints at all, it will reposition the cursor to the beginning of the table.

Note: Ensure that you update the values of the search parameters in the buffers before you call this function; the new values are read during this function call.

Limitations

1. SaCursorSearchReset() can not be used in the following scenarios:
   - The search has a local sort, i.e. not all sorting criteria could be solved by the index used for the search
   - The search is done by rowid with SaCursorSearchByRowid

   In these cases, SaCursorSearchReset returns SA_ERR_NORESETSEARCH.
2. Each "like" value that you use in constraints must be the same length. The reason for this is that SaCursorLike() takes the length of the "like" constraint as an argument, but it is not possible to change this length when SaCursorSearchReset() is called. For example, the function will work correctly if you use the following sequence of "like" values, because they are all the same length:

   `"SMITH"`

   `"JONES"`

   However, the function will not work correctly if you use the following sequence of "like" values:

   `"SMITH"`

   `"JOHNSON"`
3. Using SaCursorSearchReset is usually impractical if you set multiple constraints using the same column binding. For example, suppose that you want to search for values of "col" in the range between 1 and 10 (inclusive). Your code would look like the following example::

   ```
   SaCursorColInt(scur, "col", &i);
   i = 1;
   SaCursorAtleast(scur, "col");
   i = 10;
   SaCursorAtmost(scur, "col");
   ```

   If you reset a search like this, the new value for the column is read from the variable i only once. Therefore, the server reads one value and uses it as both the upper and lower bound. For example, suppose that you use the following code:

   ```
   i = 5;
   SaCursorSearchReset(scur);
   ```

   This code makes the search 5 <= i <= 5, which is not the desired result.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorSearchReset(
    SaCursorT* scur
```

The SaCursorSearchReset function accepts the following parameters:

Table 96. SaCursorSearchReset Parameters

Parameters	**Usage Type**	**Description**
scur	in, use	Pointer to a cursor object

Return Value

SA_RC_SUCC or error code.

SaCursorSetLockMode

SaCursorSetLockMode sets the cursor search mode.

This setting affects the possible locking modes in the server. If a search is already active, the setting will affect only the next search done in the same cursor. By default the search mode is SA_LOCK_SHARE.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorSetLockMode(
    SaCursorT* scur,
    sa_lockmode_t lockmode)
```

The SaCursorSetLockMode function accepts the following parameters:

Table 97. SaCursorSetLockMode Parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
lockmode	in	Search mode that can be one of the following mode: • SA_LOCK_SHARE • SA_LOCK_FORUPDATE • SA_LOCK_EXCLUSIVE

The meanings of the various modes are as follows:

- SA_LOCK_SHARE: default optimistic concurrency control.
- SA_LOCK_FORUPDATE: locks the row for update; others can only read, not write.
- SA_LOCK_EXCLUSIVE: locks the row exclusively; others cannot read or write this record.

Note: This function applies to any table; the table does not need to have a particular lock mode for this function to apply.

Return Value

SA_RC_SUCC

SA_ERR_ILLENUMVAL

SaCursorSetPosition

SaCursorSetPosition positions the cursor to a row specified by a key value. The key value is taken from user bound column variables which have a constraint specification.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorSetPosition(
    SaCursorT* scur)
```

The SaCursorSetPosition function accepts the following parameters:

Table 98. SaCursorSetPosition parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Return value

SA_RC_SUCC or error code.

SaCursorSetRowsPerMessage

SaCursorSetRowsPerMessage sets the number of rows to be sent in one network message from the server to the client.

The setting has no effect after the search has been started by function SaCursorSearch.

Synopsis

```
SaRetT SA_EXPORT_H
    SaCursorSetRowsPerMessage(
    SaCursorT* scur,
    int rows_per_message)
```

The SaCursorSetRowsPerMessage function accepts the following parameters:

Table 99. SaCursorSetRowsPerMessage parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object
rows_per_message	in	Number of rows to send in one network message

Return value

SA_RC_SUCC Success

SA_ERR_FAILED Error, rows_per_message < 1

SaCursorUpdate

SaCursorUpdate updates the current row in a cursor in the database.

The cursor must be positioned to a row. Column values for the new row are taken from the user variables bound to columns.

Synopsis

```
SaRetT SA_EXPORT_H SaCursorUpdate(SaCursorT* scur)
```

The SaCursorUpdate function accepts the following parameters:

Table 100. SaCursorUpdate parameters

Parameters	Usage Type	Description
scur	in, use	Pointer to a cursor object

Return value

SA_RC_SUCC or error code.

SaDateCreate

SaDateCreate creates a new date object.

The date stored in the date object is undefined.

Synopsis

```
SaDateT* SA_EXPORT_H SaDateCreate(void)
```

The SaDateCreate function accepts no parameters.

Return value

Table 101. SaDateCreate Return Values

Return Usage Type	Description
give	A new date object

SaDateFree

SaDateFree releases a date object.

After this call, the date object is invalid and cannot be used.

Synopsis

```
void SA_EXPORT_H SaDateFree(SaDateT* date)
```

The SaDateFree function accepts the following parameters:

Table 102. SaDateFree parameters

Parameters	Usage Type	Description
date	in, take	Date object

Return value

None.

SaDateSetAsciiz

SaDateSetAsciiz sets ASCII zero string date to a date object.

The following special characters are recognized in the format string:

```
YYYY        year including century
YY          year with default century 1900
MM          month
M           month
DD          day of month
D           day of month
HH          hours
H           hours
NN          minutes
N           minutes
SS          seconds
S           seconds
FFF         fractions of a second, 1/1000 seconds
```

All fields are optional. The fields are scanned from the format string, and when a match is found, the field is replaced with the proper value. All other characters in the format are treated literally.

Double letters (for example, "MM", "DD", etc.) indicate that the values should be expressed with two digits (values 1-9 will be preceded with the 0 character, for example, 01). Single letters indicate that the value should be expressed with one digit if possible. For example, if you define the date format as "YY-M-D" then the date January 2, 1999 will look like "99-1-2". If you define the date format as "YY-MM-DD", then the date will look like "99-01-02"

The following examples show the usage of date formats:

```
SaDateSetAsciiz(date, "YY-MM-DD", "94-09-13");
```

```
SaDateSetAsciiz(date, "MM/DD/YY HH.NN", "09/13/94 19.20");
```

The default date format is YYYY-MM-DD HH:NN:SS, where time fields are optional.

Synopsis

```
SaRetT SA_EXPORT_H SaDateSetAsciiz(
    SaDateT* date,
    char* format,
    char* asciiz)
```

The SaDateSetAsciiz function accepts the following parameters:

Table 103. SaDateSetAsciiz Parameters

Parameters	Usage Type	Description
date	in, out	Date object
format	in, use	Format of date in asciiz (zero-terminated ASCII) buffer, or NULL if default format is used
asciiz	in, use	Buffer containing the data in asciiz (zero-terminated ASCII) string format

Return Value

SA_RC_SUCC

SA_ERR_FAILED

SaDateSetTimet

SaDateSetTimet copies the input value from the variable named "timet" to the variable named "date". The value is automatically converted from the format time_t (the format returned by C-library function time()) to the format SaDateT.

Synopsis

```
SaRetT SA_EXPORT_H SaDateSetTimet(
SaDateT* date,
long timet)
```

The SaDateSetTimet function accepts the following parameters:

Table 104. SaDateSetTimet parameters

Parameters	Usage Type	Description
date	use	Date object
timet	in	New date value in time_t format

Return value

SA_RC_SUCC

SA_ERR_FAILED

SaDateToAsciiz

SaDateToAsciiz stores the date in an ASCII zero-terminated string format.

For an explanation of different date formats, see “SaDateSetAsciiz” on page 135.

Synopsis

```
SaRetT SA_EXPORT_H SaDateToAsciiz(
SaDateT* date,
char* format,
char* asciiz)
```

The SaDateToAsciiz function accepts the following parameters:

Table 105. SaDateToAsciiz parameters

Parameters	Usage Type	Description
date	in, use	Date object
format	in, use	Format of date in asciiz (zero-terminated ASCII) buffer, or NULL if default format is used
asciiz	out	Buffer where date is stored. **Note:** Note that the caller must allocate a sufficiently large buffer before calling this function, and is also responsible for deallocating the buffer when done with it.

Return Value

SA_RC_SUCC

SA_ERR_FAILED

SaDateToTimet

SaDateToTimet stores the date in a time_t format. The time_t date is the same value as returned by C-library function time().

Synopsis

```
SaRetT SA_EXPORT_H SaDateToTimet(
SaDateT* date,
long* p_timet)
```

The SaDateToTimet function accepts the following parameters:

Table 106. SaDateToTimet parameters

Parameters	Usage Type	Description
date	in, use	Date object
timet	out	Pointer to a long variable into where the date is stored in time_t format.

Return value

SA_RC_SUCC

SA_ERR_FAILED

SaDefineChSet

SaDefineChSet defines the client character set.

Synopsis

```
SaRetT SA_EXPORT_H SaDefineChSet(
SaConnectT* scon,
SaChSetT chset)
```

The SaDefineChSet function accepts the following parameters:

Table 107. SaDefineChSet parameters

Parameters	Usage Type	Description
scon	in out	Pointer to a connection object
chset	in	Enumerated charset specification. The valid character sets are listed in the `sa.h` file and include SA_CHSET_DEFAULT, SA_CHSET_ANSI, etc.

Note: The usage type of scon includes "out" because the scon parameter is modified by this function call.

Return value

SA_RC_SUCC when OK or SA_ERR_CHSETUNSUPP when specified character set is not supported.

SaDfloatCmp

SaDfloatCmp compares two dfloat values.

Synopsis

```
int SA_EXPORT_H SaDfloatCmp(
SaDfloatT* p_dfl1,
SaDfloatT* p_dfl2)
```

The SaDfloatCmp function accepts the following parameters:

Table 108. SaDfloatCmp parameters

Parameters	Usage Type	Description
p_dfl1	in, use	Pointer to dfloat variable
p_dfl2	in, use	Pointer to dfloat variable

Return value

```
<   -1 if p_dfl1 < p_dfl2
=   0  if p_dfl1 = p_dfl2
>   1 if p_dfl1 > p_dfl2
```

This parallels the strcmp() function in C, which returns a negative number if the first parameter is less than the second, zero if the two are equal, and a positive number (greater than zero) if the first parameter is greater than the second.

SaDfloatDiff

SaDfloatDiff calculates the difference of two dfloat values (that is, p_dfl1 - p_dfl2). The result is stored into * p_result_dfl.

Synopsis

```
SaRetT SA_EXPORT_H SaDfloatDiff(
    SaDfloatT* p_result_dfl,
    SaDfloatT* p_dfl1,
    SaDfloatT* p_dfl2)
```

The SaDfloatDiff function accepts the following parameters:

Table 109. SaDfloatDiff parameters

Parameters	Usage Type	Description
p_result_dfl	out	Pointer to dfloat variable where the result is stored.
p_dfl1	in, use	Pointer to dfloat variable

Table 109. SaDfloatDiff parameters (continued)

Parameters	Usage Type	Description
p_dfl2	in, use	Pointer to dfloat variable

Return value

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatOverflow

SaDfloatOverflow checks if the dfloat contains an overflow value.

Synopsis

```
int SA_EXPORT_H SaDfloatOverflow(
    SaDfloatT* p_dfl)
```

The SaDfloatOverflow function accepts the following parameters:

Table 110. SaDfloatOverflow parameters

Parameters	Usage Type	Description
p_dfl	in, use	Pointer to dfloat variable

Return value

1: dfloat value is an overflow value

0: dfloat value is not an overflow value

SaDfloatProd

SaDfloatProd calculates the product of two dfloat values. The result is stored into * p_result_dfl.

Synopsis

```
SaRetT SA_EXPORT_H SaDfloatProd(
SaDfloatT* p_result_dfl,
SaDfloatT* p_dfl1,
SaDfloatT* p_dfl2)
```

The SaDfloatProd function accepts the following parameters:

Table 111. SaDfloatProd Parameters

Parameters	Usage Type	Description
p_result_dfl	out	Pointer to dfloat variable where the result is stored.
p_dfl1	in	Pointer to dfloat variable.

Table 111. SaDfloatProd Parameters (continued)

Parameters	Usage Type	Description
p_dfl2	in	Pointer to dfloat variable.

Return Value

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatQuot

SaDfloatQuot calculates the quotient of two dfloat values (that is, p_dfl1 / p_dfl2). The result is stored into * p_result_dfl.

Synopsis

```
SaRetT SA_EXPORT_H SaDfloatQuot(
    SaDfloatT* p_result_dfl,
    SaDfloatT* p_dfl1,
    SaDfloatT* p_dfl2)
```

The SaDfloatQuot function accepts the following parameters:

Table 112. SaDfloatQuot parameters

Parameters	Usage Type	Description
p_result_dfl	out	Pointer to dfloat variable where the result is stored
p_dfl1	in	Pointer to dfloat variable.
p_dfl2	in	Pointer to dfloat variable.

Return value

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatSetAsciiz

SaDfloatSetAsciiz sets the value of the dfloat from a zero-terminated ASCII string.

Synopsis

```
SaRetT SA_EXPORT_H SaDfloatSetAsciiz(
    SaDfloatT* p_dfl,
    char* asciiz)
```

The SaDfloatSetAsciiz function accepts the following parameters:

Table 113. SaDfloatSetAsciiz parameters

Parameters	Usage Type	Description
p_dfl1	out	Pointer to dfloat variable where the result is stored.
asciiz	in	Buffer where the dfloat value is read as a zero-terminated ASCII string.

Return value

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatSum

SaDfloatSum calculates the sum of two dfloat values. The result is stored into * p_result_dfl.

Synopsis

```
SaRetT SA_EXPORT_H SaDfloatSum(
    SaDfloatT* p_result_dfl,
    SaDfloatT* p_dfl1,
    SaDfloatT* p_dfl2)
```

The SaDfloatSum function accepts the following parameters:

Table 114. SaDfloatSum parameters

Parameters	Usage Type	Description
p_result_dfl	out	Pointer to dfloat variable where the result is stored
p_dfl1	in	Pointer to dfloat variable.
p_dfl2	in	Pointer to dfloat variable.

Return value

SA_RC_SUCC or error code.

SaDfloatToAsciiz

SaDfloatToAsciiz stores the dfloat value as an asciiz (zero-terminated ASCII) string.

Synopsis

```
SaRetT SA_EXPORT_H SaDfloatToAsciiz(
    SaDfloatT* p_dfl,
    char* asciiz)
```

The SaDfloatToAsciiz function accepts the following parameters:

Table 115. SaDfloatToAsciiz parameters

Parameters	Usage Type	Description
p_dfl	in	Pointer to dfloat variable.
asciiz	out	Buffer where the dfloat is stored in asciiz (zero-terminated ASCII) string format. The memory for this must already be allocated by the caller.

Return value

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatUnderflow

SaDfloatUnderflow checks if the dfloat contains an underflow value.

Synopsis

```
int SA_EXPORT_H SaDfloatUnderflow(
    SaDfloatT* p_dfl)
```

The SaDfloatUnderflow function accepts the following parameters:

Table 116. SaDfloatUnderflow parameters

Parameters	Usage Type	Description
p_dfl	in, use	Pointer to dfloat variable.

Return value

1: dfloat value is an underflow value

0: dfloat value is not an underflow value

SaDisconnect

SaDisconnect disconnects the user from the solidDB server.

Synopsis

```
void SA_EXPORT_H SaDisconnect(SaConnectT* scon)
```

The SaDisconnect function accepts the following parameters:

Table 117. SaDisconnect Parameters

Parameters	Usage Type	Description
scon	in, take	Pointer to a connection object.

Return Value

None.

SaDynDataAppend

SaDynDataAppend appends data to the dynamic data object.

Synopsis

```
void SA_EXPORT_H SaDynDataAppend(
    SaDynDataT* dd,
    char* data,
    unsigned len)
```

The SaDynDataAppend function accepts the following parameters:

Table 118. SaDynDataAppend parameters

Parameters	Usage Type	Description
dd	use	Dynamic data object.
data	in out, use	Data that is appended to the dd.
len	in	Length of the data to be appended.

Return Value

None.

See also

See "SaCursorColDynData" on page 115 for a more detailed discussion of binding variables.

SaDynDataChLen

SaDynDataChLen changes the data area length of dynamic data object. It allocates and deallocates memory as necessary.

If the new length is smaller than the current length, the data area is truncated. If the new length is greater than the current length, the new data area content is initialized with space characters.

Synopsis

```
void SA_EXPORT_H SaDynDataChLen(
    SaDynDataT* dd,
    unsigned len)
```

The SaDynDataChLen function accepts the following parameters:

Table 119. SaDynDataChLen Parameters

Parameters	Usage Type	Description
dd	in out, use	Dynamic data object.
len	in	New data area length of dynamic data object.

Return Value

None.

See Also

See “SaCursorColDynData” on page 115 for a more detailed discussion of "Dynamic Data" (SaDynDataT).

SaDynDataClear

SaDynDataClear deallocates the memory allocations from an SaDynDataT object.

SaDynDataClear deallocates the data; it does not deallocate the SaDynDataT object itself. The result of SaDynDataClear leaves an "empty" dynamic data object returned by SaDynDataCreate. The SaDynDataT object itself must be deallocated separately using the SaDynDataFree function.

Synopsis

```
void SA_EXPORT_H SaDynDataClear(
    SaDynDataT* dd)
```

The SaDynDataClear function accepts the following parameters:

Table 120. SaDynDataClear Parameters

Parameters	Usage Type	Description
dd	in out, use	Dynamic data object.

Return Value

None.

See Also

See “SaCursorColDynData” on page 115 for a more detailed discussion of "Dynamic Data" (SaDynDataT).

SaDynDataCreate

SaDynDataCreate creates a new dynamic data object. A dynamic data object is an object that can hold variable amounts of any type of data.

Dynamic data objects can be manipulated using other SaDynDataXXX functions.

Synopsis

```
SaDynDataT* SA_EXPORT_H SaDynDataCreate(void)
```

SaDynDataCreate accepts no parameters.

Return Value

Table 121. SaDynDataCreate Return Value

Return Usage Type	Description
give	A new empty dynamic data object. Returns NULL in case of an error.

See Also

See “SaCursorColDynData” on page 115 for a more detailed discussion of "Dynamic Data" (SaDynDataT).

SaDynDataFree

SaDynDataFree releases a dynamic data object. After this call, the dynamic data object pointer is invalid and cannot be used.

Synopsis

```
void SA_EXPORT_H SaDynDataFree(
    SaDynDataT* dd)
```

The SaDynDataFree function accepts the following parameters:

Table 122. SaDynDataFree parameters

Parameters	Usage Type	Description
dd	in, take	Dynamic data object.

Return Value

None

See also

See “SaCursorColDynData” on page 115 for a more detailed discussion of "Dynamic Data" (SaDynDataT).

SaDynDataGetData

SaDynDataGetData returns the pointer to the data area of the dynamic data object.

Synopsis

```
char* SA_EXPORT_H SaDynDataGetData(
    SaDynDataT* dd)
```

The SaDynDataGetData function accepts the following parameters:

Table 123. SaDynDataGetData parameters

Parameters	Usage Type	Description
dd	in, use	Dynamic data object.

Return Value

A reference to the local data area of dynamic data object.

See also

See “SaCursorColDynData” on page 115 for a more detailed discussion of "Dynamic Data" (SaDynDataT).

SaDynDataGetLen

SaDynDataGetLen returns the length of the data area of the dynamic data object.

Synopsis

```
unsigned SA_EXPORT_H SaDynDataGetLen(
    SaDynDataT* dd)
```

The SaDynDataGetData function accepts the following parameters:

Table 124. SaDynDataGetData Parameters

Parameters	Usage Type	Description
dd	in, use	Dynamic data object.

Return Value

Data area length. The function returns 0 if there is an error, or if the actual length of the data area is 0.

See Also

See “SaCursorColDynData” on page 115 for a more detailed discussion of "Dynamic Data" (SaDynDataT).

SaDynDataMove

SaDynDataMove copies data from the parameter named "data" to a dynamic data object (named dd). This function overwrites possible existing data.

The parameter dd must point to a Dynamic Data Object previously created with the SaDynDataCreate function.

Note: SaDynDataMove copies the data. To copy just the reference rather than the data, see “SaDynDataMoveRef” on page 148.

Typically, the functions SaDynDataMove and SaDynDataAppend are used to set and modify the data value inside the dynamic data object. More memory will be automatically allocated when necessary and all the associated memory will be automatically deallocated when the dynamic data object is disposed of using SaDynDataFree. The user can access the data or the length using the respective functions SaDynDataGetData and SaDynDataGetLen.

The use of SaDynDataMove and SaDynDataAppend may not be feasible when the data already exists completely in a memory buffer. In addition to increasing the memory usage by keeping two copies of the same data, the overhead of the memory copy may be significant if the buffers are large. Therefore, it may be wise to directly assign the data pointer by using SaDynDataMoveRef (rather than copying by using SaDynDataMove). In this case, the user may modify or deallocate the memory buffer only after the dynamic data object itself has been freed.

Synopsis

```
void SA_EXPORT_H SaDynDataMove(
SaDynDataT* dd,
char* data,
unsigned len)
```

The SaDynDataMove function accepts the following parameters:

Table 125. SaDynDataMove Parameters

Parameters	**Usage Type**	**Description**
dd	in out, use	Dynamic data object.
data	in, use	New data
len	in	Length of data (if the data is a string, this length should include the string terminator)

Return Value

None.

See Also

“SaDynDataMoveRef.”

See “SaCursorColDynData” on page 115 for more information about "Dynamic Data" (SaDynDataT).

SaDynDataMoveRef

SaDynDataMoveRef moves a data reference to a dynamic dataobject.

SaDynDataMoveRef copies the pointer (address) from the parameter named "data" to the appropriate field of the parameter named "dd". The caller must guarantee that the input data is alive as long as the dynamic data object refers to that data.

Note: This function copies only the reference, not the data. To copy the data rather than just the reference, see “SaDynDataMove” on page 147.

Typically, the functions SaDynDataMove and SaDynDataAppend are used to set and modify the data value inside the dynamic data object. More memory will be automatically allocated when necessary and all the associated memory will be automatically deallocated when the dynamic data object is disposed of using SaDynDataFree. The user can access the data or the length using the respective functions SaDynDataGetData and SaDynDataGetLen.

The use of SaDynDataMove and SaDynDataAppend may not be feasible when the data already exists completely in a memory buffer. In addition to increasing the memory usage by keeping two copies of the same data, the overhead of the memory copy may be significant if the buffers are large. Therefore, it may be better to directly assign the data pointer by using SaDynDataMoveRef (rather than copying by using SaDynDataMove). In this case, the user may modify or deallocate the memory buffer only after the dynamic data object itself has been freed.

Synopsis

```
void SA_EXPORT_H SaDynDataMoveRef(
    SaDynDataT* dd,
    char* data,
    unsigned len)
```

The SaDynDataMoveRef function accepts the following parameters:

Table 126. SaDynDataMoveRef Parameters

Parameters	Usage Type	Description
dd	in out, use	Dynamic data object
data	in, hold	Data
len	in	Length of data (if the data is a string, this length should include the string terminator)

Return Value

None

See Also

"SaDynDataMove" on page 147.

See "SaCursorColDynData" on page 115 for a more detailed discussion of "Dynamic Data" (SaDynDataT).

SaDynStrAppend

SaDynStrAppend appends another string at the end of a dynamic string.

Synopsis

```
void SA_EXPORT_H SaDynStrAppend(
    SaDynStrT* p_ds,
    char* str)
```

The SaDynStrAppend function accepts the following parameters:

Table 127. SaDynStrAppend parameters

Parameters	Usage Type	Description
p_ds	out	Dynamic string
str	in, use	String that is appended at p_ds

Return Value

None.

SaDynStrCreate

SaDynStrCreate creates (initializes) a new dynamic string object.

Synopsis

```
SaDynStrT SA_EXPORT_H SaDynStrCreate(void)
```

SaDynDataGetData accepts no parameters.

Return Value

Table 128. SaDynStrCreate Return Value

Return Usage Type	Description
give	Dynamic string object initialized with empty data. Returns NULL if running out of memory.

SaDynStrFree

SaDynStrFree frees the SaDynStrT variable.

In search operations, the column data is stored to the SaDynStrT variable using function SaDynStrMove, which overwrites the old data. The user is responsible for releasing the SaDynStrT variable after the search ends using function SaDynStrFree.

Synopsis

```
void SA_EXPORT_H SaDynStrFree(
    SaDynStrT* p_ds)
```

The SaDynStrFree function accepts the following parameters:

Table 129. SaDynStrFree parameters

Parameters	Usage Type	Description
p_ds	in, take	Dynamic string.

Note: Because the function deallocates the memory, the pointer p_ds is no longer valid after the function call and thus the usage type is "take".

Return Value

None.

SaDynStrMove

SaDynStrMove copies the value of the string (the second parameter) to the SaDynStrT (the first parameter).

SaDynStrMove copies the string, not the pointer.

The SaDynStrT must be initialized with SaDynStrCreate before SaDynStrT is set with SaDynStrMove.

CAUTION:
Do not copy a SaDynStrT to another SaDynStrT (for example with memcpy). This would result in two SaDynStrT pointers pointing at the same allocated area.

Synopsis

```
void SA_EXPORT_H SaDynStrMove(
    SaDynStrT* p_ds,
    char* str)
```

The SaDynStrMove function accepts the following parameters:

Table 130. SaDynStrMove Parameters

Parameters	Usage Type	Description
p_ds	out	Pointer to a dynamic string variable.
str	in, use	New value of a dynamic string.

Return Value

None.

SaErrorInfo

SaErrorInfo returns error information from the last operation in a server connection.

Cursor errors cannot be checked with this function; instead function SaCursorErrorInfo must be used.

Synopsis

```
bool SA_EXPORT_H SaErrorInfo(
SaConnectT* scon,
char** errstr,
int* errcode)
```

The SaErrorInfo function accepts the following parameters:

Table 131. SaErrorInfo Parameters

Parameters	Usage Type	Description
scon	use	Pointer to a connection object.
errstr	out, ref	If there was an error, and if this parameter is non-NULL, then a pointer to a local copy of an error string is stored into *errstr.
errcode	out	If there was an error, and if this parameter is non-NULL, then an error code is stored into *errcode.

Return Value

TRUE There was an error, so errstr and errcode were updated.

FALSE There were no errors, so errstr and errcode were not updated.

SaGlobalInit

SaGlobalInit performs some global initialization in the SA system.

This function must be called before any other SA function except SaConnect. If the SaConnect function is called before any other SA function, then you do not need to call SaGlobalInit because SaConnect will call it for you.

Synopsis

```
void SA_EXPORT_H SaGlobalInit(void)
```

SaGlobalInit accepts no parameters.

Return Value

None.

SaSetDateFormat

SaSetDateFormat defines default date format.

For explanation of the possible date formats, see "SaDateToAsciiz" on page 137.

Synopsis

```
SaRetT SA_EXPORT_H SaSetDateFormat(
SaConnectT* scon,
char* dateformat)
```

The SaSetDateFormat function accepts the following parameters:

Table 132. SaSetDateFormat Parameters

Parameters	Usage Type	Description
scon	in, out, use	Pointer to a connection object.

Table 132. SaSetDateFormat Parameters (continued)

Parameters	Usage Type	Description
dateformat	in, use	Default date format for connection.

Note: The usage type includes "out" because the scon parameter is modified by this function call.

Return Value

SA_RC_SUCC if success.

SA_ERR_COMERROR if the connection to the server is broken.

See Also

For explanation of the possible date, time, and timestamp formats, see "SaDateToAsciiz" on page 137.

SaSetSortBufSize

SaSetSortBufSize sets the amount of memory that a connection uses for local sorts (sorts that are done on the client side by the SA library).

Synopsis

```
SaRetT SA_EXPORT_H SaSetSortBufSize(
    SaConnectT* scon,
    unsigned long size)
```

The SaSetSortBufSize function accepts the following parameters:

Table 133. SaSetSortBufSize Parameters

Parameters	Usage Type	Description
scon	in, out, use	Pointer to a connection object.
size	in	Memory buffer size in bytes.

Note: The usage type includes "out" because the scon parameter is modified by this function call.

Return Value

SA_RC_SUCC when OK or SA_ERR_FAILED when specified memory size was too small (< 10KB)

SaSetSortMaxFiles

SaSetSortMaxFiles sets the maximum number of files that the connection uses for local sorts (sorts that are done on the client side by the SA library).

Synopsis

```
SaRetT SA_EXPORT_H SaSetSortMaxFiles(
SaConnectT* scon,
unsigned int nfiles)
```

The SaSetSortMaxFiles function accepts the following parameters:

Table 134. SaSetSortMaxFiles parameters

Parameters	Usage Type	Description
scon	in, out, use	Pointer to a connection object.
nfiles	in	Maximum number of files

Note: The usage type includes "out" because the scon parameter is modified by this function call.

Return Value

SA_RC_SUCC when OK or SA_ERR_FAILED when specified number of files is too small (< 3).

SaSetTimeFormat

SaSetTimeFormat defines the default time format.

For an explanation of the possible formats, see the time portion documentation of SaDateSetAsciiz in "SaDateSetAsciiz" on page 135.

Synopsis

```
SaRetT SA_EXPORT_H SaSetTimeFormat(
    SaConnectT* scon,
    char* timeformat)
```

The SaSetTimeFormat function accepts the following parameters:

Table 135. SaSetTimeFormat Parameters

Parameters	Usage Type	Description
scon	in, out, use	Pointer to a connection object.
timeformat	in	Default time format for connection.

Note: The usage type of scon includes "out" because the scon parameter is modified by this function call.

Return Value

SA_RC_SUCC

SA_ERR_COMERROR if the connection to the server is broken.

See Also

For an explanation of the possible date, time, and timestamp formats, see "SaDateSetAsciiz" on page 135.

SaSetTimestampFormat

SaSetTimestampFormat defines the default timestamp format.

For an explanation of the possible date, time, and timestamp formats, see "SaDateSetAsciiz" on page 135.

Synopsis

```
SaRetT SA_EXPORT_H SaSetTimestampFormat(
    SaConnectT* scon,
    char* timestampformat)
```

The SaSetTimestampFormat function accepts the following parameters:

Table 136. SaSetTimestampFormat parameters

Parameters	Usage Type	Description
scon	in, out, use	Pointer to a connection object.
timestampformat	in	Default timestamp format for connection.

Return Value

SA_RC_SUCC

See Also

For an explanation of the possible date, time, and timestamp formats, see "SaDateSetAsciiz" on page 135.

SaSQLExecDirect

SaSQLExecDirect allows you to execute simple SQL statements such as CREATE TABLE, DROP TABLE, INSERT, and DELETE.

You cannot do SELECT operations because it is not possible to fetch the data.

Synopsis

```
SaRetT SA_EXPORT_H SaSQLExecDirect(SaConnectT* scon,
char *sqlstr)
```

The SaSQLExecDirect function accepts the following parameters:

Table 137. SaSQLExecDirect Parameters

Parameters	Usage Type	Description
scon	in, use	Pointer to a connection object.

Table 137. SaSQLExecDirect Parameters (continued)

Parameters	Usage Type	Description
sqlstr	in, use	Pointer to a string containing the SQL statement to execute.

Return Value

SA_RC_SUCC

The possible error codes are as follows:
- 15001: SAP_ERR_SYNTAXERROR_SD. Syntax error: <error>, <line>.
- 15002: SAP_ERR_ILLCOLNAME_S. Illegal column name <name>.
- 15003: SAP_ERR_TOOMANYPARAMS. Too many parameters for string constraints.
- 15004: SAP_ERR_TOOFEWPARAMS. Too few parameters for string constraints.

SaTransBegin

SaTransBegin starts a new transaction. After this call, all select, insert, update and delete operations are executed in the same transaction, and the changes are not visible in the database until SaTransCommit is called.

Without the SaTransBegin call, the server is in autocommit mode by default and therefore each select, insert, update, and delete operation is executed in a separate transaction. No explicit commit (SaTransCommit) is required when in autocommit mode.

The transaction is run in a mode where write operations are validated for lost updates and unique errors.

Synopsis

```
void SA_EXPORT_H SaTransBegin(SaConnectT* scon)
```

The SaTransBegin function accepts the following parameters:

Table 138. SaTransBegin parameters

Parameters	Usage Type	Description
scon	in, use	Pointer to a connection object.

Return Value

None.

SaTransCommit

SaTransCommit commits the current transaction started by SaTransBegin.

After calling this function, all changes are made persistent in the database. After the current transaction is completed, the database server returns to autocommit mode until the next call to SaTransBegin.

Synopsis

```
SaRetT SA_EXPORT_H SaTransCommit(SaConnectT* scon)
```

The SaTransCommit function accepts the following parameters:

Table 139. SaTransCommit Parameters

Parameters	Usage Type	Description
scon	in, use	Pointer to a connection object

Return Value

SA_RC_SUCC or error code.

SaTransRollback

SaTransRollback rolls back the current transaction started by SaTransBegin. No changes are made to the database.

After the current transaction is completed, the database server returns to autocommit mode until the next call to SaTransBegin.

Synopsis

```
SaRetT SA_EXPORT_H SaTransRollback(SaConnectT* scon)
```

The SaTransRollback function accepts the following parameters:

Table 140. SaTransRollback Parameters

Parameters	Usage Type	Description
scon	in, use	Pointer to a connection object

Return Value

SA_RC_SUCC or error code.

SaUserId

SaUserID returns current user id of a connection.

Synopsis

```
int SA_EXPORT_H SaUserId(SaConnectT* scon)
```

The SaUserId function accepts the following parameters:

Table 141. SaUserId Parameters

Parameters	**Usage Type**	**Description**
scon	in, use	Pointer to a connection object

Return Value

User id in the server.

Appendix A. solidDB supported ODBC functions

This topic describes the ODBC functions supported by solidDB.

Table 142. solidDB supported ODBC functions

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
Connecting to a Data Source			
SQLAllocEnv (1.0)	N/A	Deprecated (replaced by SQLAllocHandle)	N/A
SQLAllocConnect (1.0)	N/A	Deprecated (replaced by SQLAllocHandle)	N/A
SQLAllocHandle (3.0)	Returns the list of supported data source attributes. Returns the list of installed drivers and their attributes.	Supported Supported	ISO 92 ODBC
SQLConnect (1.0)	Establishes connections to a driver and a data source. The connection handle references storage of all information about the connection to the data source, including status, transaction state, and error information.	Supported	ISO 92
SQLDriverConnect (1.0)	This function is an alternative to SQLConnect. It supports data sources that require more connection information than the three arguments in SQLConnect, including dialog boxes to prompt the user for all connection information, and data sources that are not defined in the system information.	Supported (including Unicode version of this function).	ODBC
* Version introduced is the version when the function was initially added to the ODBC API. ** Conformance level can be ISO 92 (also appears in X/Open version 1 because X/Open is a pure superset of ISO 92), X/Open (also appears in ODBC 3.x because ODBC 3.x is a pure superset of X/Open version 1), ODBC (appears in neither ISO 92 or X/Open) or N/A (Deprecated in ODBC 3.x).			
SQLBrowseConnect (1.0)	Returns successive levels of attributes and attribute values. When all levels have been enumerated, a connection to the data source is completed and a complete connection string is returned. A return of SQL_SUCCESS_WITH_INFO indicates that all connection information has been specified and the application is now connected to the data source.	Not supported	ISO 92
SQLGetInfo (1.0)	Returns general information about the driver and data source associated with a connection.	Supported	ISO 92

Table 142. solidDB supported ODBC functions (continued)

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLGetFunctions (1.0)	Returns information about whether a driver supports a specific ODBC function.	Supported; this function is implemented in the ODBC Driver Manager. It can also be implemented in drivers. If a driver implements SQLGetFunctions, the Driver manager calls the function in the driver. Otherwise, it executes the function itself. In the case of solidDB, the function is implemented in the driver so that the application linked to the driver can also call this function from the application.	ISO 92
SQLGetTypeInfo (1.0)	Returns information about data types supported by the data source. The driver returns the information in the form of an SQL result set. The data types are intended for use in Data Definition Language (DDL) statements.	Supported	ISO 92
Obtaining Information about a Driver and Data Source			
SQLDataSources (1.0)	Returns information about a data source.	Supported; this function is implemented in the ODBC Driver Manager. For non-Microsoft Windows platforms which do not have the Microsoft ODBC Driver manager, this function is not supported.	ISO 92
SQLDrivers (2.0)	Lists driver descriptions and driver attribute keywords.	Supported; this function is implemented in the ODBC Driver Manager. For Microsoft Windows, the Driver Manager is required if applications that connect to solidDB use OLE DB or ADO APIs or if database tools that require the Driver Manager, such as Microsoft Access, FoxPro, or Crystal Reports are to be used. For platforms other than Microsoft Windows, the Driver Managers are provided by vendors such as iODBC, Merant, and UnixODBC, etc.	ODBC
SQLGetConnectAttr (3.0)	Returns the value of a connection attribute.	Supported	ISO 92
SQLSetConnectAttr (3.0)	Sets a connection attribute.	Supported	ISO 92

Table 142. solidDB supported ODBC functions (continued)

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLGetEnvAttr (3.0) SQLSetEnvAttr (3.0)	Returns the value of an environment attribute. Sets an environment attribute.	Supported Supported	ISO 92 ISO 92
SQLGetStmtAttr (3.0) SQLSetStmtAttr (3.0)	Returns the value of a statement attribute. Sets a statement attribute.	Supported Supported	ISO 92 ISO 92
SQLSetConnectOption (1.0) SQLGetConnectOption (1.0)	N/A	Deprecated (replaced by SQLSetConnectAttr) Deprecated (replaced by SQLGetConnectAttr)	N/A N/A
SQLGetStmtOption (1.0) SQLSetStmtOption (1.0)	N/A N/A	Deprecated (replaced by SQLGetStmtAttr) Deprecated (replaced by SQLSetStmtAttr)	N/A N/A
Setting and Retrieving Descriptor Fields			
SQLGetDescField (3.0) SQLSetDescField (3.0)	Returns the current setting or value of a single descriptor field. Sets the value of a single field of a descriptor record.	Supported Supported	ISO 92 ISO 92
SQLGetDescRec (3.0) SQLSetDescRec (3.0)	Returns the current settings or values of multiple fields of a descriptor record. The fields returned describe the name, data type, and storage of column or parameter data. Sets multiple descriptor fields that affect the data type and buffer bound to a column or parameter data.	Supported Supported	ISO 92 ISO 92
SQLCopyDesc (3.0)	Copies descriptor information from one descriptor handle to another.	Supported	ISO 92
Preparing SQL Requests			
SQLAllocStmt (1.0)	N/A	Deprecated (replaced by SQLAllocHandle)	N/A
SQLPrepare (1.0)	Prepares an SQL statement for later execution.	Supported	ISO 92
SQLBindParameter (2.0)	Assigns storage for a parameter in an SQL statement.	Supported Note: This function replaces SQLBindParam which did not exist in ODBC 2.x, although it is in the X/Open and ISO standards.	ODBC

Table 142. solidDB supported ODBC functions (continued)

Function Names/Version Introduced*	**Purpose**	**Availability when using ODBC**	**Conformance****
SQLGetCursorName (1.0) SQLSetCursorName (1.0)	Returns the cursor name associated with a statement handle. Specifies a cursor name with an active statement. If an application does not call SQLSetCursorName, the driver generates cursor names as needed for SQL statement processing.	Supported Supported	ISO 92 ISO 92
SQLParamOptions (1.0)	N/A	Deprecated (replaced by SQLSetStmtAttr)	N/A
SQLSetParam (1.0)	N/A	Deprecated (replaced by SQLBindParameter)	N/A
SQLSetScrollOptions (1.0)	Sets options that control cursor behavior.	Deprecated (replaced by SQLGetInfo and SQLSetStmtAttr)	ODBC
	Submitting Requests		
SQLExecute (1.0)	Executes a prepared statement using the current values of the parameter marker variables if any parameter markers exist in the statement.	Supported	ISO 92
SQLExecDirect (1.0)	Executes a preparable statement using the current values of the parameter marker variables if any parameters exist in the statement. SQLExecDirect is the fastest way to submit an SQL statement for one-time execution.	Supported	ISO 92
SQLNativeSQL (1.0)	Returns the SQL string as modified by the driver. SQLNativeSQL does not execute the SQL statement.	Not implemented; solidDB does not support this functionality.	N/A
SQLDescribeParam (1.0)	Returns the text of an SQL statement as translated by the driver. This information is also available in the fields of the IPD.	Supported	ODBC
SQLNumParams (1.0)	Returns the number of parameters in an SQL statement.	Supported	ISO 92
SQLParamData (1.0)	Used in conjunction with SQLPutData to supply parameter data at execution time. (Useful for long data values.)	Supported	ISO 92
SQLPutData (1.0)	Allows an application to send data for a parameter or column to the driver at statement execution time. This function can be used to send character or binary data values in parts to a column with a character, binary, or data source-specific data type (for example, parameters of the SQL_LONGVARBINARY or SQL_LONGVARCHAR types).	Supported	ISO 92

Table 142. solidDB supported ODBC functions (continued)

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
	Retrieving Results and Information about Results		
SQLRowCount (1.0)	Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement.	Supported	ISO 92
SQLNumResultCols (1.0)	Returns the number of columns in a result set.	Supported	ISO 92
SQLDescribeCol (1.0)	Returns the result descriptor (column name, type, column size, decimal digits, and nullability) for one column in the result set. This information is also available in the fields of the IRD. NOTE: The driver now returns the number of characters instead of the number of bytes for the following attributes: SQL_DESC_LABEL, SQL_DESC_NAME, SQL_DESC_SCHEMA_ NAME, SQL_DESC_CATALOG_ NAME, SQL_DESC_BASE_COLUMN_NAME, and SQLDESC_BASE_TABLE_NAME This conforms more closely to the ODBC standard and works correctly using ADO, VB, OLE-DB, and ODBC calls. Note, however, that this causes failure of the Microsoft Visual DataBase Project. After updating/inserting the record, the record is not saved and the following error is displayed: "the table does not exist.	Supported.	ISO 92
SQLColAttributes (1.0)	N/A	Deprecated (replaced by SQLColAttribute)	N/A
SQLColAttribute (3.0)	Describes attributes of a column in the result set. **Note:** The driver now returns the number of characters instead of the number of bytes for the following attributes: SQL_DESC_LABEL, SQL_DESC_NAME, SQL_DESC_SCHEMA_ NAME, SQL_DESC_CATALOG_NAME, SQL_DESC_BASE_COLUMN_NAME, and SQLDESC_BASE_TABLE_NAME This conforms more closely to the ODBC standard and works correctly using ADO, VB, OLE-DB, and ODBC calls. Note, however, that this causes failure of the Microsoft Visual DataBase Project. After updating/inserting the record, the record is not saved and the following error is displayed: "the table does not exist."	Supported.	ISO 92
SQLBindCol (1.0)	Assigns storage for a result column and specifies the data type.	Supported	ISO 92
SQLFetch (1.0)	Returns multiple result rows, fetching the next rowset of data from the result set and returning data for all bound columns.	Supported	ISO 92

Table 142. solidDB supported ODBC functions (continued)

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLExtendedFetch (2.0)	N/A	Replaced by SQLFetchScroll	N/A
SQLFetchScroll (3.0)	Returns scrollable result rows, fetching the specified rowset of data from the result set and returning data for all bound columns. Block cursor support enables an application to fetch more than one row with a single fetch into the application buffer. When working with an ODBC 2.x driver, the Driver Manager maps this function to SQLExtendedFetch.	Supported **Note:** Since the solidDB ODBC Driver currently has no support for bookmarks, it is not possible to support the SQL_FETCH_BOOKMARK option in SQLFetchScroll.	ISO 92
SQLGetData (1.0)	Returns part or all of one column of one row of a result set. It can be called multiple times to retrieve variable length data in parts, making it useful for long data values.	Supported	ISO 92
SQLSetPos (1.0)	Positions a cursor within a fetched block of data and allows an application to refresh data in the rowset or to update or delete data in the result set.	Supported, along with all the options, that is, SQL_POSITION, SQL_DELETE, and SQL_UPDATE	ODBC
SQLBulkOperations (3.0)	Performs bulk insertions and bulk bookmark operations, including update, delete, and fetch by bookmark.	solidDB supports this, but only when using the SQL_ADD option.	ODBC
SQLMoreResults (1.0)	Determines whether there are more results available on a statement containing SELECT, UPDATE, INSERT, or DELETE statement and, if so, initializes processing for those results.	Not supported solidDB does not support multiple results.	ODBC
SQLGetDiagField (3.0)	Returns additional diagnostic information (a single field of the diagnostic data structure associated with a specified handle). This information includes error, warning, and status information.	Supported	ISO 92
SQLGetDiagRec (3.0)	Returns additional diagnostic information (multiple fields of the diagnostic data structure). Unlike SQLGetDiagField, which returns one diagnostic field per call, SQLGetDiagRec returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the diagnostic message text.	Supported	ISO 92
SQLError (1.0)	N/A	Deprecated (replaced by SQLGetDiagRec)	N/A
Obtaining Information about the Data Source's System Tables			

Table 142. solidDB supported ODBC functions (continued)

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLColumnPrivileges (1.0)	Returns a list of columns and associated privileges for the specified table. The driver returns the information as a result set on the specified StatementHandle. This function is supported via an appropriate SQL execution.	Supported	ODBC
SQLColumns (1.0)	Returns a list of columns and associated privileges for the specified table. The driver returns the information as a result set on the specified StatementHandle. This function is supported via an appropriate SQL execution.	Supported	X/Open
SQLForeignKeys (1.0)	Returns two type of lists: • Foreign keys in the specified table (columns in the specified table that refer to primary keys in other tables). • Foreign keys in other tables that refer to the primary key in the specified table. The driver returns each list as a result set on the specified statement.	Supported	ODBC
SQLPrimaryKeys (1.0)	Returns the list of column names that make up the primary key for a table. The driver returns the information as a result set. This function does not support returning primary keys from multiple tables in a single call.	Supported	ODBC
SQLProcedureColumns (1.0)	Returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures. The driver returns the information as a result set on the specified statement.	Supported.	ODBC
SQLProcedures (1.0)	Returns the list of procedure names stored in a specific data source. Procedure is a generic term used to describe an executable object, or a named entity that can be invoked using input and output parameters.	Supported	ODBC
SQLSpecialColumns (1.0)	Returns the following information about columns within a specified table: • The optimal set of columns that uniquely identifies a row in the table. • Columns that are automatically updated when any value in the row is updated by a transaction.	Supported	X/Open
SQLStatistics (1.0)	Returns statistics about a single table and the list of indexes associated with the table. The driver returns the information as a result set.	Supported	ISO 92

Table 142. solidDB supported ODBC functions (continued)

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLTablePrivileges (1.0)	Returns a list of tables and the privileges associated with each table. The driver returns the information as a result set on the specified statement.	Supported	ODBC
SQLTables (1.0)	Returns the list of table, catalog, or schema names, and table types, stored in a specific data source.	Supported	X/Open
Terminating a statement			
SQLFreeStmt (1.0)	Ends statement processing, discards pending results, and optionally, frees all resources associated with the statement handle.	Supported Note: The SQLFreeStmt with an option of SQL_DROP is replaced by SQLFreeHandle.	ISO 92
SQLCloseCursor (3.0)	Closes a cursor that has been opened on a statement, and discards pending results.	Supported	ISO 92
SQLCancel (1.0)	Cancels the processing on an SQL statement.	Supported	ISO 92
SQLEndTran (3.0)	Requests a transaction commit or rollback on all statements associated with a connection. SQLEndTran can also request that a commit or rollback operation be performed for all connections associated with an environment.	Supported	ISO 92
SQLTransact (1.0)	N/A	Deprecated (replaced by SQLEndTran)	N/A
Terminating a Connection			
SQLDisconnect (1.0)	Closes the connection associated with a specific connection handle.	Supported	ISO 92
SQLFreeConnect (1.0)	N/A	Deprecated (replaced by SQLFreeHandle)	N/A
SQLFreeEnv (1.0)	N/A	Deprecated (replaced by SQLFreeHandle)	N/A
SQLFreeHandle (3.0)	Frees resources associated with a specific environment, connection, statement, or descriptor handle	Supported	ISO 92

Appendix B. solidDB ODBC Driver 3.5.1 Attributes Support

This topic provides information about the solidDB ODBC Driver 3.5.1 attributes.

The attributes are grouped in the following categories:

- Environment-level attributes
- Connection-level attributes
- Statement-level attributes
- Column-level attributes

Table 143. 001 Environment Level

Attribute	Value (Option)	Driver Manager	Driver Alone	Comments
SQL_ATTR_ CONNECTION_ POOLING	SQL_CP_OFF SQL_CP_ ONE_PER_DRIVER SQL_CP_ ONE_PER_HENV	All values are supported	All values are not applicable to the driver	All values are not applicable to ODBC drivers, handled by Driver Manager, so this attribute will be supported if the application links to ODBC DM and can't be simulated by the driver itself.
SQL_ATTR_CP_MATCH	SQL_CP_ STRICT_MATCH SQL_CP_ RELAXED_MATCH	All values supported	All values are not applicable to the driver	All values are not applicable to ODBC drivers, handled by Driver Manager, so this attribute will be supported if the application links to ODBC DM and can't be simulated by the driver itself.
SQL_ATTR_ ODBC_VERSION	SQL_OV_ODBC3 SQL_OV_ODBC2	Supported Not Supported	Supported Not Supported	Allows user to set and get the version to 2, but the behavior is a per 3.0 and above.
SQL_ATTR_ OUTPUT_NTS	SQL_TRUE SQL_FALSE	Supported Not Supported	Supported Not Supported	

Table 144. 002 Connection Level

Attribute	Value (Option)	Driver Manager	Driver Alone	Comments
SQL_ATTR_ODBC_CURSORS	SQL_CUR_IF_NEEDED SQL_FETCH_PRIOR SQL_CUR_USE_ODBC SQL_CUR_ USE_DRIVER	All values not supported	All values not supported	
SQL_ATTR_ACCESS_MODE	SQL_MODE_ READ_ONLY SQL_MODE_ READ_WRITE	All values not supported	All values not supported	
SQL_ATTR_ASYNC_ENABLE	SQL_ASYNC_ ENABLE_OFF SQL_ASYNC_ ENABLE_ON	All values not supported	All values not supported	
SQL_ATTR_AUTO_IPD	SQL_TRUE SQL_FALSE	All values not supported	All values not supported	
SQL_ATTR_AUTOCOMMIT	SQL_ATTR_ AUTOCOMMIT_OFF SQL_ATTR_ AUTOCOMMIT_ON	All values are supported	All values are supported	
SQL_ATTR_ CONNECTION_ TIMEOUT	Timeout value in sec	Supported	Supported	
SQL_ATTR_ CURRENT_CATALOG	CatalogName	Supported	Supported	
SQL_ATTR_ LOGIN_TIMEOUT	Timeout value in sec	Supported	Supported	
SQL_ATTR_ METADATA_ID	SQL_TRUE SQL_FALSE	All values not supported	All values not supported	
SQL_ATTR_PACKET_SIZE	Packet size in bytes	the desired size	Not supported	
SQL_ATTR_QUIET_MODE	Set to NULL	can set and get	Not supported	
SQL_ATTR_TRACE	SQL_TRACE_OFF SQL_TRACE_ON	All values supported	All values not supported	All values handled by DM, not by driver

Table 144. 002 Connection Level (continued)

Attribute	Value (Option)	Driver Manager	Driver Alone	Comments
SQL_ATTR_TRACEFILE	Pointer to trace file name	Supported	Not supported	Handled by DM, not by driver
SQL_ATTR_TRNSLATE_LIB	Pointer Name of lib	Supported	Not supported	Handled by DM, not by driver
SQL_ATTR_TXN_ISOLATION	SQL_TXN_ SERIALIZABLE SQL_TXN_READ_ UNCOMMITTED SQL_TXN_READ_ COMMITTED SQL_TXN_ REPEATABLE_ READ	All values are supported, except SQL_TXN_ READ_ UNCOMMITTED	All values are supported, except SQL_TXN_ READ_ UNCOMMITTED	An solidDB server does not support the READ_ UNCOMMITTED feature.

Table 145. 03 Statement Level

Attribute	Value (Option)	Driver Manager	Driver Alone	Comments
SQL_ATTR_ CONCURRENCY	SQL_CONCUR_ READ_ONLY SQL_CONCUR_LOCK SQL_CONCUR_ROWVER SQL_CONCUR_VALUES	All values supported	All values supported	For the value SQL_CONCUR_READ_ONLY, set and get are supported. For all other values, set is supported and get returns READ_ONLY.
SQL_ATTR_ CURSOR_TYPE	SQL_CURSOR_ FORWARD_ONLY SQL_CURSOR_ KEYSET_DRIVEN SQL_CURSOR_DYNAMIC SQL_CURSOR_STATIC	Supported Forced to dynamic Forced to dynamic Forced to dynamic	Supported Forced to dynamic Forced to dynamic Forced to dynamic	
SQL_ATTR_MAX_LENGTH	Length in bytes	Not supported	Not supported	Whatever the length, sets only to default (0).
SQL_ATTR_MAX_ROWS	Maximum number of rows	Not supported	Not supported	Whatever the length, sets only to default (0).
SQL_ATTR_ RETRIEVE_DATA	SQL_RD_OFF SQL_RD_ON	Not supported Supported	Not supported Supported	Sets to SQL_RD_ON only

Table 145. 03 Statement Level (continued)

Attribute	Value (Option)	Driver Manager	Driver Alone	Comments
SQL_ATTR_ USE_BOOKMARKS	SQL_UB_OFF SQL_UB_ON	All values not supported	All values not supported	
SQL_ATTR_ ROW_ARRAY_SIZE	An SQLUSMALLINT* value that points to an array of SQLUSMALLINT values containing row status values after a call to SQLFetch or SQLFetchScroll.	Supported	Supported	The array has as many elements as there are rows in the rowset.
SQL_ATTR_ROWS_ FETCHED_PTR	An SQLUINTEGER* value that points to a buffer in which to return the number of rows fetched after a call to SQLFetch or SQLFetchScroll.	Supported	Supported	
SQL_ATTR_ ROW_STATUS_PTR	An SQLUINTEGER value that specifies the number of rows returned by each call to SQLFetch or SQLFetchScroll.	Supported	Supported	
SQL_ROWSET_SIZE	Number of rows to return	Supported	Supported	Allows the ODBC application to set its value to greater than 1.
SQL_ASYNC_ENABLE	SQL_ASYNC_ ENABLE_ON SQL_ASYNC_ ENABLE_OFF	All values not supported	All values not supported	
SQL_BIND_TYPE	SQL_BIND_BY_COLUMN	Not supported	Not supported	
SQL_ATTR_KEYSET_SIZE	Size	Not supported	Not supported	Whatever the size, sets only to default (0).
SQL_ATTR_NOSCAN	SQL_NOSCAN_OFF SQL_NOSCAN_ON	Not supported Not supported	Not supported Not supported	Sets to SQL_NOSCAN_OFF only
SQL_ATTR_ SIMULATE_CURSOR	SQL_SC_NON_UNIQUE SQL_SC_TRY_UNIQUE SQL_SC_UNIQUE	All values not supported	All values not supported	All values are not relevant to solidDB Driver

Table 145. 03 Statement Level (continued)

Attribute	Value (Option)	Driver Manager	Driver Alone	Comments
SQL_ATTR_ APP_PARAM_DESC	SQL_NULL_HDESC	Not supported	Not supported	
SQL_ATTR_ APP_ROW_DESC	SQL_NULL_HDESC	Not supported	Not supported	
SQL_ATTR_ CURSOR_SCROLLABLE	SQL_SCROLLABLE SQL_NONSCROLLABLE	Not supported Not supported	Not supported Not supported	Sets to SQL_NONSCROLLABLE only
SQL_ATTR_ CURSOR_SENSITIVITY	SQL_UNSPECIFIED SQL_INSENSITIVE SQL_SENSITIVE	Not supported Not supported Not supported	Not supported Not supported Not supported	Sets to SQL_UNSPECIFIED only Sets to SQL_ UNSPECIFIED only
SQL_ATTR_ROW_NUMBER	Number of current row	Supported	Supported	User can get the number of rows; cannot set because of read-only property
SQL_ATTR_ ENABLE_AUTO_IPD	SQL_TRUE SQL_FALSE	Both values not supported	Both values not supported	
SQL_ATTR_METADATA_ID	SQL_TRUE SQL_FALSE	Both values not supported	Both values not supported	
SQL_ATTR_PARAM_ BIND_OFFSET_PTR	SQL_DESC_DATA_PTR SQL_DESC_ INDICATOR_PTR SQL_DESC_ OCTET_LENGTH_PTR SQL_DESC_ BIND_OFFSET_PTR	All values are supported	All values are supported	
SQL_ATTR_PARAM_ OPERATION_PTR	Pointer to array containing list of parameters to be ignored	Supported	Supported	

Table 145. 03 Statement Level (continued)

Attribute	Value (Option)	Driver Manager	Driver Alone	Comments
SQL_ATTR_PARAMS_ PROCESSED_PTR	Unsigned integer pointer to return the number of sets of parameters that have been processed by the SQL statement executed through SQLExecute or SQLExecDirect.	Supported	Supported	

Table 146. 04 Column Attributes

Attribute	Value (Option)	Driver Manager	Driver Alone	Comments
SQL_DESC_BASE_COLUMN_NAME		Supported	Supported	
SQL_DESC_BASE_TABLE_NAME		Supported	Supported	
SQL_DESC_DISPLAY_SIZE		Supported	Supported	
SQL_DESC_NAME				
SQL_DESC_NULLABLE		Supported	Supported	
SQL_DESC_OCTET_LENGTH		Supported	Supported	
SQL_DESC_PRECISION		Supported	Supported	
SQL_DESC_SCALE		Supported	Supported	
SQL_DESC_UPDATABLE		Supported	Supported	
SQL_DESC_FIXED_PREC_SCALE		Supported	Supported	
SQL_DESC_TABLE_NAME		Supported	Supported	
SQL_DESC_TYPE		Supported	Supported	
SQL_DESC_UNNAMED		Supported	Supported	
SQL_DESC_SCHEMA_NAME		Supported	Supported	
SQL_DESC_LOCAL_TYPE_NAME		Supported	Supported	
SQL_DESC_LABEL		Supported	Supported	
SQL_DESC_TYPE_NAME		Supported	Supported	
SQL_DESC_AUTO_UNIQUE_VALUE		Supported	Supported	

Table 146. 04 Column Attributes (continued)

Attribute	Value (Option)	Driver Manager	Driver Alone	Comments
SQL_DESC_CONCISE_TYPE		Supported	Supported	
SQL_DESC_LITERAL_PREFIX		Supported	Supported	
SQL_DESC_UNSIGNED		Supported	Supported	
SQL_DESC_LITERAL_PREFIX		Supported	Supported	
SQL_DESC_UNSIGNED		Supported	Supported	
SQL_DESC_LITERAL_SUFFIX		Supported	Supported	
SQL_DESC_CATALOG_NAME		Supported	Supported	
SQL_DESC_COUNT		Supported	Supported	
SQL_DESC_SEARCHABLE		Supported	Supported	
SQL_DESC_LENGTH		Supported	Supported	
SQL_DESC_CASE_SENSITIVE		Supported	Supported	
SQL_DESC_NUM_PREX_RADIX		Supported	Supported	

Appendix C. Error codes

This topic contains an error codes table that provides possible SQLSTATE values that a driver returns for the SQLGetDiagRec function.

Note: The SQLGetDiagRec and SQLGetDiagField return SQLSTATE values that conform to the X/Open (The Open Group) *Data Management: Structured Query Language (SQL), Version 2 (3/95)*.

Error Codes Table Convention

Table 147. Error Code Class Values

Class value	Meaning
01	Indicates a warning and includes a return code of SQL_SUCCESS_WITH_INFO. **Note:** Error class 01 returns both warnings and errors.
01, 07, 08, 21, 22, 23, 24, 25, 28, 34, 3C, 3D, 3F, 40, 42, 44, HY	Indicates an error that includes a return value of SQL_ERROR. **Note:** Error class 01 returns both warnings and errors.
IM	Indicates warning and errors that are derived from ODBC.

Note: Typically, when a function successfully executes, it returns a value of SQL_SUCCESS; in some cases, however, the function may also return the SQLSTATE 00000, which also indicates successful execution.

SQLSTATE codes

Table 148. SQLSTATE codes

SQLSTATE	Error	Can be returned from
01000	General warning	All ODBC functions except: SQLGetDiagField SQLGetDiagRec
01001	Cursor operation conflict	SQLExecDirect SQLExecute SQLParamData SQLSetPos
01002	Disconnect error	SQLDisconnect

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
01003	NULL value eliminated in set function	SQLExecDirect SQLExecute SQLParamData
01004	String data, right truncated	• SQLColAttribute • SQLDataSources • SQLDescribeCol • SQLDriverConnect • SQLDrivers • SQLExecDirect • SQLExecute • SQLExtendedFetch • SQLFetch • SQLFetchScroll • SQLGetConnectAttr • SQLGetCursorName • SQLGetData • SQLGetDescField • SQLGetDescRec • SQLGetEnvAttr • SQLGetInfo • SQLGetStmtAttr • SQLParamData • SQLPutData • SQLSetCursorName
01006	Privilege not revoked	SQLExecDirect SQLExecute SQLParamData
01007	Privilege not granted	SQLExecDirect SQLExecute SQLParamData
01S00	Invalid connection string attribute	SQLDriverConnect SQLSetPos
01S01	Error in row	SQLExtendedFetch
01S02	Option value changed	• SQLConnect • SQLDriverConnect • SQLExecDirect • SQLExecute • SQLParamData • SQLPrepare • SQLSetConnectAttr • SQLSetDescField • SQLSetEnvAttr • SQLSetStmtAttr

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
01S06	Attempt to fetch before the result set returned the first rowset	SQLExtendedFetch SQLFetchScroll
01S07	Fractional truncation	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLSetPos
01S08	Error saving File DSN	SQCriverConnect
01S09	Invalid keyword	SQLDriverConnect
07001	Wrong number of parameters	SQLExecDirect SQLExecute
07002	COUNT field incorrect	SQLExecDirect SQLExecute SQLParamData
07005	Prepared statement not a cursor_specification	SQLColAttribute SQLDescribeCol
07006	Restricted data type attribute violation	SQLBindCol SQLBindParameter SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
07009	Invalid descriptor index	SQLBindCol SQLBindParameter SQLColAttribute SQLDescribeCol SQLDesribeParam SQLFetch SQLFetchScroll SQLGetData SQLGetDescField SQLParamData SQLSetDescField SQLSetDescRec SQLSetPos
07S01	Invalid use of default parameter	SQLExecDirect SQLExecute SQLParamData SQLPutData
08001	Client unable to establish connection	SQLConnect SQLDriverConnect
08002	Connection name in use	SQLConnect SQLDriverConnect SQLSetConnectAttr
08003	Connection does not exist	SQLAllocHandle SQLDisconnect SQLEndTran SQLGetConnectAttr SQLGetInfo SQLSetConnectAttr
08004	Server rejected the connection	SQLConnect SQLDriverConnect
08007	Connection failure during transaction	SQLEndTran

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
08S01	Communication link failure	• SQLColumnPrivileges • SQLColumns • SQLConnect • SQLConnect • SQLCopyDesc • SQLDescribeCol • SQLDescribeParam • SQLDriverConnect • SQLExecDirect • SQLExecute • SQLExtendedFetch • SQLFetch • SQLFetchScroll • SQLForeignKeys • SQLGetConnectAttr • SQLGetData • SQLGetDescField • SQLGetDescRec • SQLGetFunctions • SQLGetInfo • SQLGetTypeInfo • SQLMoreResults • SQLNumParams • SQLNumResultCols • SQLParamData • SQLPrepare • SQLPrimaryKeys • SQLProcedureColumns • SQLProcedures • SQLPutData • SQLSetConnectAttr • SQLSetDescField • SQLSetDescRec • SQLSetEnvAttr • SQLSetStmtAttr • SQLSpecialColumns • SQLStatistics • SQLTablePrivileges • SQLTables
21S01	Insert value list does not match column list	SQLExecDirect SQLPrepare
21S02	Degree of derived table does not match column list	SQLExecDirect SQLExecute SQLParamData SQLPrepare SQLSetPos

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
22001	String data, right truncated	SQLExecDirect SQLExecute SQLFetch SQLFetchScroll SQLParamData SQLPutData SQLSetDescField SQLSetPos
22002	Indicator variable required but not supplied	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData
22003	Numeric value out of range	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLGetInfo SQLParamData SQLPutData SQLSetPos
22007	Invalid datetime format	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData SQLSetPos

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
22008	Datetime field overflow	SQLExecDirect SQLExecute SQLParamData SQLPutData
22012	Division by zero	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData
22015	Interval field overflow	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData SQLSetPos
22018	Invalid character value for cast specification	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData SQLSetPos
22019	Invalid escape character	SQLExecDirect SQLExecute SQLPrepare

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
22025	Invalid escape sequence	SQLExecDirect SQLExecute SQLPrepare
22026	String data, length mismatch	SQLParamData
23000	Integrity constraint violation	SQLExecDirect SQLExecute SQLParamData SQLSetPos
24000	Invalid cursor state	SQLCloseCursor SQLColumnPrivileges SQLColumns SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLForeignKeys SQLGetData SQLGetStmtAttr SQLGetTypeInfo SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLConnectAttr SQLSetCursorName SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
25000	Invalid transaction state	SQLDisconnect
25S01	Transaction state	SQLEndTran
25S02	Transaction is still active	SQLEndTran

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
25S03	Transaction is rolled back	SQLEndTran
28000	Invalid authorization specification	SQLConnect SQLDriverConnect
34000	Invalid cursor name	SQLExecDirect SQLPrepare SQLSetCursorName
3C000	Duplicate cursor name	SQLSetCursorName
3D000	Invalid catalog name	SQLExecDirect SQLPrepare SQLSetConnectAttr
3F000	Invalid schema name	SQLExecDirect SQLPrepare
40001	Serialization failure	SQLColumnPrivileges SQLColumns SQLEndTran SQLExecDirect SQLExecute SQLFetch SQLFetchScroll SQLForeignKeys SQLGetTypeInfo SQLMoreResults SQLParamData SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
40002	Integrity constraint violation	SQLEndTran

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
40003	Statement completion unknown	SQLColumnPrivileges SQLColumns SQLExecDirect SQLExecute SQLFetch SQLFetchScroll SQLGetTypeInfo SQLForeignKeys SQLMoreResults SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLParamData SQLSetPos SQLSpecialColumns SQLStatistics SQLTables
42000	Syntax error or access violation	SQLExecDirect SQLExecute SQLParamData SQLPrepare SQLSetPos
42S01	Base table or view already exists	SQLExecDirect SQLPrepare
42S02	Base table or view not found	SQLExecDirect SQLPrepare
42S11	Index already exists	SQLExecDirect SQLPrepare
42S12	Index not found	SQLExecDirect SQLPrepare
42S21	Column already exists	SQLExecDirect SQLPrepare
42S22	Column not found	SQLExecDirect SQLPrepare

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
44000	WITH CHECK OPTION violation	SQLExecDirect SQLExecute SQLParamData
HY000	General Error	All ODBC functions except: SQLGetDiagField SQLGetDiagRec
HY001	Memory allocation error	All ODBC function except: SQLGetDiagField SQLGetDiagRec
HY003	Invalid application buffer type	SQLBindCol SQLBindParameter SQLGetData
HY004	Invalid SQL data type	SQLBindParameter SQLGetTypeInfo
HY007	Associated statement is not prepared	SQLCopyDesc SQLGetDescField SQLGetDescRec

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
HY008	Operation canceled	All ODBC functions that can be processed asynchronously: SQLColAttribute SQLColumnPrivileges SQLColumns SQLDescribeCol SQLDescribeParam SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLForeignKeys SQLGetData SQLGetTypeInfo SQLMoreResults SQLNumParams SQLNumResultCols SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
HY009	Invalid use of null pointer	SQLAllocHandle
		SQLBindParameter
		SQLColumnPrivileges
		SQLColumns
		SQLExecDirect
		SQLForeignKeys
		SQLGetCursorName
		SQLGetData
		SQLGetFunctions
		SQLPrepare
		SQLPrimaryKeys
		SQLProcedureColumns
		SQLProcedures
		SQLPutData
		SQLSetConnectAttr
		SQLSetCursorName
		SQLSetEnvAttr
		SQLSetStmtAttr
		SQLSpecialColumns
		SQLStatistics
		SQLTablePrivileges
		SQLTables

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
HY010	Function sequence error	• SQLAllocHandle • SQLBindCol • SQLBindParameter • SQLCloseCursor • SQLColAttribute • SQLColumnPrivileges • SQLColumns • SQLCopyDesc • SQLDescribeCol • SQLDescribeParam • SQLDisconnect • SQLEndTran • SQLExecDirect • SQLExecute • SQLExtendedFetch • SQLFetch • SQLFetchScroll • SQLForeignKeys • SQLFreeHandle • SQLFreeStmt • SQLGetConnectAttr • SQLGetCursorName • SQLGetData • SQLGetDescField • SQLGetDescRec • SQLGetFunctions • SQLGetStmtAttr • SQLGetTypeInfo • SQLMoreResults • SQLNumParams • SQLNumResultCols • SQLParamData • SQLPrepare • SQLPrimaryKeys • SQLProcedureColumns • SQLProcedures • SQLPutData • SQLRowCount • SQLSetConnectAttr • SQLSetCursorName • SQLSetDescField • SQLSetEnvAttr • SQLSetDescRec • SQLSetPos • SQLSetStmtAttr • SQLSpecialColumns • SQLStatistics • SQLTablePrivileges • SQLTables

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
HY011	Attribute cannot be set now	SQLParamData SQLSetConnectAttr SQLSetPos SQLSetStmtAttr
HY012	Invalid transaction operation code	SQLEndTran
HY013	Memory Management err	All ODBC functions except: SQLGetDiagField SQLGetDiagRec
HY014	Limit on the number of handles exceeded	SQLAllocHandle
HY015	No cursor name available	SQLGetCursorName
HY016	Cannot modify an implementation row descriptor	SQLCopyDesc SQLSetDescField SQLSetDescRec
HY017	Invalid use of an automatically allocated descriptor handle	SQLFreeHandle SQLSetStmtAttr
HY018	Server declined cancel request	SQLCancel
HY019	Non-character and non-binary data sent in pieces	SQLPutData
HY020	Attempt to concatenate a null value	SQLPutData
HY021	Inconsistent descriptor information	SQLBindParameter SQLCopyDesc SQLGetDescField SQLSetDescField SQLSetDescRec
HY024	Invalid attribute value	SQLSetConnectAttr SQLSetEnvAttr SQLSetStmtAttr

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
HY090	Invalid string or buffer length	• SQLBindCol • SQLBindParameter • SQLBrowseConnect • SQLColAttribute • SQLColumnPrivileges • SQLColumns • SQLConnect • SQLDataSources • SQLDescribeCol • SQLDriverConnect • SQLDrivers • SQLExecDirect • SQLExecute • SQLFetch • SQLFetchScroll • SQLForeignKeys • SQLGetConnectAttr • SQLGetCursorName • SQLGetData • SQLGetDescField • SQLGetInfo • SQLGetStmtAttr • SQLParamData • SQLPrepare • SQLPrimaryKeys • SQLProcedureColumns • SQLProcedures • SQLPutData • SQLSetConnectAttr • SQLSetCursorName • SQLSetDescField • SQLSetDescRec • SQLSetEnvAttr • SQLSetStmtAttr • SQLSetPos • SQLSpecialColumns • SQLTablePrivileges • SQLStatistics • SQLTables
HY091	Invalid descriptor field identifier	SQLColAttribute SQLGetDescField SQLSetDescField

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
HY092	Invalid attribute/option identifier	SQLAllocHandle SQLCopyDesc SQLDriverConnect SQLEndTran SQLFreeStmt SQLGetConnectAttr SQLGetEnvAttr SQLGetStmtAttr SQLParamData SQLSetConnectAttr SQLSetDescField SQLSetEnvAttr SQLSetPos SQLSetStmtAttr
HY095	Function type out of range	SQLGetFunctions
HY096	Invalid information type	SQLGetInfo
HY097	Column type out of range	SQLSpecial Columns
HY098	Scope type out of range	SQLSpecial Columns
HY099	Nullable type out of range	SQLSpecial Columns
HY100	Uniqueness option type out of range	SQLStatistics
HY101	Accuracy option type out of range	SQLStatistics
HY103	Invalid retrieval code	SQLDataSources SQLDrivers
HY104	Invalid precision or scale value	SQLBindParameter
HY105	Invalid parameter type	SQLBindParameter SQLExecDirect SQLExecute SQLParamData SQLSetDescField
HY106	Fetch type out of range	SQLExtendedFetch SQLFetchScroll

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
HY107	Row value out of range	SQLExtendedFetch SQLFetch SQLFetchScroll SQLSetPos
HY109	Invalid cursor position	SQLExecDirect SQLExecute SQLGetData SQLGetStmtAttr SQLParamData SQLSetPos
HY110	Invalid driver completion	SQLDriverConnect
HY111	Invalid bookmark value	SQLExtendedFetch SQLFetchScroll

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
HYC00	Optional feature not implemented	• SQLBindCol • SQLBindParameter • SQLColAttribute • SQLColumnPrivileges • SQLColumns • SQLDriverConnect • SQLEndTran • SQLConnect • SQLExecDirect • SQLExecute • SQLExtendedFetch • SQLFetch • SQLFetchScroll • SQLForeignKeys • SQLGetConnectAttr • SQLGetData • SQLGetEnvAttr • SQLSetPos • SQLGetInfo • SQLGetStmtAttr • SQLGetTypeInfo • SQLParamData • SQLPrepare • SQLPrimaryKeys • SQLProcedureColumns • SQLProcedures • SQLSetConnectAttr • SQLSetEnvAttr • SQLSetStmtAttr • SQLSpecialColumns • SQLStatistics • SQLTablePrivileges • SQLTables

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
HYT00	Timeout expired	SQLBrowseConnect SQLColumnPrivileges SQLColumns SQLConnect SQLDriverConnect SQLExecDirect SQLExecute SQLExtendedFetch SQLForeignKeys SQLGetTypeInfo SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
HYT01	Connection timeout expired	All ODBC functions except: SQLDrivers SQLDataSources SQLGetEnvAttr SQLSetEnvAttr
IM001	Driver does not support this function	All ODBC functions except: SQLAllocHandle SQLDataSources SQLDrivers SQLFreeHandle SQLGetFunctions
IM002	Data source name not found and no default driver specified	SQLConnect SQLDriverConnect
IM003	Specified driver could not be loaded.	SQLConnect

Table 148. SQLSTATE codes (continued)

SQLSTATE	Error	Can be returned from
IM004	Driver's SQLAllocHandle on SQL_HANDLE_ENV failed	SQLDriverConnec SQLConnect SQLDriverConnect
IM005	Driver's SQLAllocHandle on SQL_HANDLE_DBC failed	SQLConnect SQLDriverConnect
IM006	Driver's SQLSetConnectAttr Failed	SQLConnect SQLDriverConnect
IM007	No data source or driver specified; dialog prohibited	SQLDriverConnect
IM008	Dialog failed	SQLDriverConnect
IM009	Unable to load translation DLL	SQLConnect SQLDriverConnect SQLSetConnectAttr
IM010	Data source name too long	SQLConnect SQLDriverConnect
IM011	Driver name too long	SQLDriverConnect
IM012	DRIVER keyword syntax error	SQLDriverConnect
IM013	Trace file error	All ODBC functions
IM014	Invalid name of File DSN	SQLDriverConnect
IM015	Corrupt file data source	SQLDriverConnect

Appendix D. Minimum SQL grammar requirements for ODBC

This section describes the minimum subset of SQL-92 Entry level syntax that an ODBC driver must support. An application that uses this syntax will be supported by any ODBC-compliant driver.

Applications can call SQLGetInfo with the SQL_SQL_CONFORMANCE to determine if additional features of SQL-92, not covered in this section, are supported.

Note: If the driver supports only read-only data sources, the SQL syntax that applies to changing data may not apply to the driver. Applications need to call SQLGetInfo with the SQL_DATA_SOURCE_READ_ONLY information type to determine if a data source is read-only.

SQL statements

This section describes the subset of SQL statements and elements.

```
create-table-statement ::=
        CREATE TABLE base_table_name
        (column_identifier data_type [, column_identifier data_type]...)
```

Important: As the *data_type* in a *create_table_statement*, applications require a data type from the TYPE_NAME column of the result set returned by SQLGetTypeInfo.

```
delete_statement_searched ::=
        DELETE FROM table_name [WHERE search_condition]
drop_table_statement ::=
        DROP TABLE base_table_name
select_statement ::=
        SELECT [ALL | DISTINCT] select_list
        FROM table_reference_list
        [WHERE search_condition]
        [order_by_clause]
statement ::= create_table_statement |
        delete_statement_searched |
        drop_table_statement |
        insert_stetement |
        select_statement |
        update_statement_searched
Update_statement_searched ::=
   UPDATE table_name
   SET column_identifier = {expression |
         NULL}
   [, column_identifier = {expression |
         NULL}]...
   [WHERE search_condition]
```

SQL statement elements

```
base_table_identifier ::= user_defined_name
base_table_name ::= base_table_identifier
boolean_factor ::= [NOT] boolean_primary
boolean_primary ::= predicate | ( search_condition )
boolean_term ::= boolean_factor [AND boolean_term]
character_string_literal :: = "{character}..."
(character is any character in the character set
of the driver/data source. To include a single
literal quote character (') in a character_string_literal,
use two literal quote characters [""].)
```

```
column_identifier ::= user_defined_name
column_name ::= [table_name.]column_identifier
comparison_operator ::= < | > | <= | >= | = | <>
comparison_predicate ::= expression comparison_operator expression
data_type ::= character_string_type
(character_string_type is any data type for which the
""DATA_TYPE"" column in the result set returned by SQLGetTypeInfo
is either SQL_CHAR or SQLVARCHAR.)
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
dynamic_parameter ::= ?
expression ::= term | expression {+|-} term
factor ::= [+|-]primary
insert_value ::= dynamic_parameter | literal | NULL | USER
letter ::= lower_case_letter | upper_case_letter
literal ::= character_string_literal
lower_case_letter ::= a | b | c | d | e | f | g |
  h | i | j | k | l | m | n | o | p | q | r | s |
  t | u | v | w | x | y | z
order_by_clause ::= ORDER BY sort_specification [, sort_specification]...
primary ::= column_name | dynamic_parameter | literal | ( expression )
search_condition ::= boolean_term [OR search_condition]
select_list ::= * | select_sublist [, select_sublist]...
(select_list cannot contain parameters.)
select_sublist ::= expression
sort_specification ::= {unsigned_integer | column_name } [ASC | DESC]
table_identifier ::= user_defined_name
table_name ::= table_identifier
table_reference ::= table_name
table_reference ::= table_name [,table_reference]...
term ::= factor | term {*|/} factor
unsigned_integer ::= {digit}
upper_case_letter ::=  A | B | C | D | E | F | G |
   H | I | J | K | L | M | N | O | P | Q | R | S |
   T | U | V | W | X | Y | Z
user_defined_name ::= letter[ digit | letter| _ ]...
```

Control statements (logical condition)

This topic provides a summary of control statements that are available in solidDB database procedures.

For a more detailed description of these control statements, see the discussion on stored procedures in the *solidDB SQL Guide*.

Table 149. Control Statements

Control statement	Description
`set variable = expression`	Assigns a value to a variable. The value can be either a literal value (e.g., 10 or 'text') or another variable. Parameters are considered as normal variables.
`variable := expression`	Alternate syntax for assigning values to variables.
`boolean_expr`	A boolean expression which evaluates to "true" or "false". The expression can include comparison operators, such as =, >, <, etc.) and logical operators and, or, and not.

Table 149. Control Statements (continued)

Control statement	Description
statement_list	A valid procedure statement that executes as a result of a boolean expression.

`while`
    boolean_expr
`loop`
    statement_list
`end loop`
	This loops while the expression is true. For examples of valid parentheses use in WHILE loops, see the discussion of stored procedures in *solidDB SQL Guide*.

`leave`	Leaves the innermost while loop and continues executing the procedure from the next statement after the keyword end loop.

`if`
    boolean_expr
`then`
    statement_list1
`else`
    statement_list2
`end if`
	Executes *statement_list1* if boolean_expr is true; otherwise, executes *statement_list2*. For examples of valid parentheses use in IF statements, see the discussion on stored procedures in *solidDB SQL Guide*.

`if`
    boolean_expr1
`then`
    statement_list1
`elseif`
    boolean_expr2
`then`
    statement_list2
`end if`
	If *boolean_expr1* is true, executes *statement_list1*. If *boolean_expr2* is true, executes *statement_list2*. The statement can optionally contain multiple elseif statements and also an else statement. For examples of valid parentheses use in IF statement, see the discussion of stored procedures in *solidDB SQL Guide*.

`return`	Returns the current values of output parameters and exits the procedure. If a procedure has a *return row* statement, *return* behaves like *return norow*.
`return sqlerror of` *cursor_name*	Returns the sqlerror associated with the cursor and exits the procedure.
`return row`	Returns the current values of output parameters and continues execution of the procedure. Return row does not exit the procedure and return control to the caller.
`return norow`	Returns the end of the set and exits the procedure.

Data type support

At minimum, ODBC drivers must support either SQL_CHAR or SQL_VARCHAR.

Other data types support is determined by the driver's or data source's SQL-92 conformance level. To determine the SQL-92 conformance level for a driver or data source, applications need to call SQLGetTypeInfo.

Parameter data types

This topic describes how data types are determined for parameters and the parameter markers support.

Even though each parameter specified with SQLBindParameter is defined using an SQL data type, the parameters in an SQL statement have no intrinsic data type. Therefore, parameter markers can be included in an SQL statement only if their data types can be inferred from another operand in the statement. For example, in an arithmetic expression such as ? + COLUMN1, the data type of the parameter can be inferred from the data type of the named column represented by COLUMN1. An application cannot use a parameter marker if the data type cannot be determined.

The following table describes how a data type is determined for several types of parameters according to SQL-92 standards. For comprehensive information on inferring the parameter type, see the SQL-92 specification.

Table 150. Determining Data Ttype for Several Types of Parameters

Location of Parameter	Assumed Data Type
One operand of a binary arithmetic or comparison operator	Same as the other operand
The first operand in a BETWEEN clause	Same as the second operand
The second or third operand in a BETWEEN clause	Same as the first operand
An expression used with IN	Same as the first value or the result column of the subquery
A value used with IN	Same as the expression or the first value if there is a parameter marker in the expression
A pattern value used with LIKE	VARCHAR
An update value used with UPDATE	Same as the update column

Parameter markers

According to the SQL-92 specification, an application cannot place parameter markers in the following locations:

- In a SELECT list.
- As both *expressions* in a *comparison-predicate*.
- As both operands of a binary operator.
- As both the first and second operands of a BETWEEN operation.
- As both the first and third operands of a BETWEEN operation.

- As both the expression and the first value of an IN operation.
- As the operand of a unary + or - operation.
- As the argument of a *set-function-reference*.

For a comprehensive list and more details, see the SQL-92 specification.

Literals in ODBC

This section contains information that will help driver writers who are converting a character string type to a numeric or interval type, or from a numeric or interval type to a character string type.

Interval literal syntax

The following syntax is used for interval literals in ODBC.

```
interval_literal ::= INTERVAL [+|_] interval_string interval_qualifier
interval_string ::= quote { year_month_literal
     | day_time_literal } quote
year_month_literal ::= years_value | [years_value] months_value
day_time_literal ::= day_time_interval | time_interval
day_time_interval ::= days_value [hours_value
     [:minutes_value[:seconds_value]]]
time_interval ::= hours_value [:minutes_value [:seconds_value ] ]
     | minutes_value [:seconds_value ]
     | seconds_value
years_value ::= datetime_value
months_value ::= datetime_value
days_value ::= datetime_value
hours_value ::= datetime_value
minutes_value ::= datetime_value
seconds_value ::= seconds_integer_value [.[seconds_fraction] ]
seconds_integer_value ::= unsigned_integer
seconds_fraction ::= unsigned_integer
datetime_value ::= unsigned_integer
interval_qualifier ::= start_field TO end_field
     | single_datetime_field
start_field ::= non_second_datetime_field
     [(interval_leading_field_precision )]
end_field ::= non_second_datetime_field
     | SECOND[(interval_fractional_seconds_precision)]
single_datetime_field ::= non_second_datetime_field
     [(interval_leading_field_precision)]
     | SECOND[(interval_leading_field_precision
     [, (interval_fractional_seconds_precision)]
datetime_field ::= non_second_datetime_field | SECOND
non_second_datetime_field ::= YEAR | MONTH | DAY | HOUR | MINUTE
interval_fractional_seconds_precision ::= unsigned_integer
interval_leading_field_precision ::= unsigned_integer
quote ::= '
unsigned_integer ::= digit...
```

Numeric Literal Syntax

The following syntax is used for numeric literals in ODBC:

```
numeric_literal ::= signed_numeric_literal | unsigned_numeric_literal
signed_numeric_literal ::= [sign] unsigned_numeric_literal
unsigned_numeric_literal ::= exact_numeric_literal
     | approximate_numeric_literal
exact_numeric_literal ::= unsigned_integer [period[unsigned_integer]]
     | period unsigned_integer
sign ::= plus_sign | minus_sign
approximate_numeric_literal ::= mantissa E exponent
```

```
mantissa ::= exact_numeric_literal
exponent ::= signed_integer
signed_integer ::= [sign] unsigned_integer
unsigned_integer ::= digit...
plus_sign ::= +
minus_sign ::= _
digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
period ::= .
```

List of reserved keywords

To ensure compatibility with drivers that support the core SQL grammar, there are keywords that applications should avoid using.

These words do not constrain the minimum SQL grammar. The #define value SQL_ODBC_KEYWORDS contains a comma-separated list of these keywords.

For a complete list of reserved keywords in several SQL standards and solidDB ODBC API, see "Reserved Words" in *IBM solidDB SQL Guide*.

Table 151. List of Reserved Keywords

Keyword	Keyword	Keyword	Keyword
ABSOLUTE	ACTION	ADA	ADD
ALL	ALLOCATE	ALTER	AND
ANY	ARE	AS	ASC
ASSERTION	AT	AUTHORIZATION	AVG
BEGIN	BETWEEN	BIT	BIT_LENGTH
BOTH	BY	CASCADE	CASCADED
CASE	CAST	CATALOG	CHAR
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CLOSE	COALESCE	COLLATE	COLLATION
COLUMN	COMMIT	CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS	CONTINUE	CONVERT
CORRESPONDING	COUNT	CREATE	CROSS
CURRENT	CURRENT_DATE	CURRENT_TIME	CURRENT_TIMESTAMP
CURRENT_USER	CURSOR	DATE	DAY
DEALLOCATE	DEC	DECIMAL	DECLARE
DEFAULT	DEFERRABLE	DEFERRED	DELETE

Table 151. List of Reserved Keywords (continued)

Keyword	**Keyword**	**Keyword**	**Keyword**
DESC	DESCRIBE	DESCRIPTOR	DIAGNOSTICS
DISCONNECT	DISTINCT	DOMAIN	DOUBLE
DROP	ELSE	END	END-EXEC
ESCAPE	EXCEPT	EXCEPTION	EXEC
EXECUTE	EXISTS	EXTERNAL	EXTRACT
FALSE	FETCH	FIRST	FLOAT
FOR	FOREIGN	FORTRAN	FOUND
FROM	FULL	GET	GLOBAL
GO	GOTO	GRANT	GROUP
HAVING	HOUR	IDENTITY	IMMEDIATE
IN	INCLUDE	INDEX	INDICATOR
INITIALLY	INNER	INPUT	INSENSITIVE
INSERT	INT	INTEGER	INTERSECT
INTERVAL	INTO	IS	ISOLATION
JOIN	KEY	LANGUAGE	LAST
LEADING	LEFT	LEVEL	LIKE
LOCAl	LOWER	MATCH	MAX
MIN	MINUTE	MODULE	MONTH
NAMES	NATIONAL	NATURAL	NCHAR
NEXT	NO	NONE	NOT
NULL	NULLIF	NUMERIC	OCTET_LENGTH
OF	ON	ONLY	OPEN
OPTION	OR	ORDER	OUTER
OUTPUT	OVERLAPS	PASCAL	POSITION
PRECISION	PREPARE	PRESERVE	PRIMARY

Table 151. List of Reserved Keywords (continued)

Keyword	Keyword	Keyword	Keyword
PRIOR	PRIVILEGES	PROCEDURE	PUBLIC
READ	REAL	REFERENCES	RELATIVE
RESTRICT	REVOKE	RIGHT	ROLLBACK
ROWS	SCHEMA	SCROLL	SECOND
SECOND	SECTION	SELECT	SESSION
SESSION_USER	SET	SIZE	SMALLINT
SOME	SPACE	SQL	SQLCA
SQLCODE	SQLERROR	SQLSTATE	SQLWARNING
SUBSTRING	SUM	SYSTEM_USER	TABLE
TEMPORARY	THEN	TIME	TIMESTAMP
TIMEZONE_HOUR	TIMEZONE_MINUTE	TO	TRAILING
TRANSACTION	TRANSLATE	TRANSLATION	TRIM
TRUE	UNION	UNIQUE	UNKNOWN
UPDATE	UPPER	USAGE	USER
USING	VALUE	VALUES	VARCHAR
VARYING	VIEW	WHEN	WHENEVER
WHERE	WITH	WORK	WRITE
YEAR	ZONE		

Appendix E. Data types

This section describes the ODBC data types.

ODBC defines the following sets of data types:

- SQL data types, which indicate the data type of data stored at the data source (e.g. the solidDB server).
- C data types, which indicate the data type of data stored in application buffers.

Each SQL data type corresponds to an ODBC C data type. Before returning data from the data source, the driver converts it to the specified C data type. Before sending data to the data source, the driver converts it from the specified C data type.

For information about driver-specific SQL data types, see the driver's documentation.

SQL data types

In accordance with the SQL-92 standard, each DBMS defines its own set of SQL data types. For each SQL data type in the SQL-92 standard, a #define value, known as a type identifier, is passed as an argument in ODBC functions or returned in the metadata of a result set.

Drivers map data source-specific SQL data types to ODBC SQL data type identifiers and driver-specific SQL data type identifiers. The SQL_DESC_CONCISE_TYPE field of an implementation descriptor is where the SQL data type is stored.

solidDB's ODBC driver does not support the following SQL_92 data types:

- BIT
- BIT_VARYING
- TIME_WITH_TIMEZONE
- TIMESTAMP_WITH_TIMEZONE
- NATIONAL_CHARACTER

C data types

ODBC defines the C data types and their corresponding ODBC type identifiers.

Applications call one of the following functions:

- SQLBindCol or SQLGetData to pass an applicable C type identifier in the TargetType argument. In this way, applications specify the C data type of the buffer that receives result set data.
- SQLBindParameter to pass the appropriate C type identifier in the ValueType argument. In this way, applications specify the C data type of the buffer containing a statement parameter.

The SQL_DESC_CONCISE_TYPE field of an application descriptor is where the C data type is stored.

Note: Driver-specific C data types do not exist.

Data type identifiers

Data type identifiers are stored in the SQL_DESC_CONCISE_TYPE field of a descriptor. Data type identifiers in applications describe their buffers to the driver.

They also retrieve metadata about the result set from the driver so applications know what type of C buffers to use for data storage. Applications use data type identifiers to perform these tasks by calling these functions:

- To describe the C data type of application buffers, applications call SQLBindParameter, SQLBindCol, and SQLGetData.
- To describe the SQL data type of dynamic parameters, applications call SQLBindParameter.
- To retrieve the SQL data types of result set columns, applications call SQLColAttribute and SQLDescribeCol.
- To retrieve the SQL data types of parameters, applications call SQLDescribeParameter.
- To retrieve the SQL data types of various schema information, applications call SQLColumns, SQLProcedureColumns, and SQLSpecialColumns.
- To retrieve a list of supported data types, applications call SQLGetTypeInfo.

In addition, the SQLSetDescField and SQLSetDescRec descriptor functions are also used to perform the above tasks. For details, see the SQLSetDescField and SQLSetDescRec functions.

SQL data types

A given driver and data source do not necessarily support all of the SQL data types defined in the ODBC grammar. Furthermore, they may support additional, driver-specific SQL data types.

A driver's support is determined by the level of SQL-92 conformance. To determine which data types a driver supports, an application calls SQLGetTypeInfo. See “SQLGetTypeInfo Result Set Example” on page 209. For information about driver-specific SQL data types, see the driver's documentation.

A driver also returns the SQL data types when it describes the data types of columns and parameters using the following functions:

- SQLColAttribute
- SQLColumns
- SQLDescribeCol
- SQLDescribeParam
- SQLProcedureColumns
- SQLSpecialColumns

Note:

For details on fields that store SQL data type values and characteristics, see “Data type identifiers and descriptors” on page 218.

The following table is not a comprehensive list of SQL data types, but offers commonly used names, ranges, and limits. A data source may only support some of the data types that are listed in the table and depending on your driver, the characteristics of the data types can differ form this table's description. The table includes the description of the associated data type from SQL-92 (if applicable)

Table 152. Common SQL Data Type Names, Ranges, and Limits

SQL Type Identifier [1]	**Typical SQL Data Type [2]**	**Typical Type Description**
SQL_CHAR	CHAR(n)	Character string of fixed string length *n*.
SQL_VARCHAR	VARCHAR(n)	Variable-length character string with a maximum string length *n*.
SQL_LONGVARCHAR	LONG VARCHAR	Variable length character data. Maximum length is data source-dependent. [3]
SQL_WCHAR	WCHAR(n)	Unicode character string of fixed string length *n*.
SQL_WVARCHAR	VARWCHAR(n)	Unicode variable-length character string with a maximum string length *n*.
SQL_WLONGVARCHAR	LONGWVARCHAR	Unicode variable-length character data. Maximum length is data source-dependent.
SQL_DECIMAL	DECIMAL(p, s)	Signed, exact, numeric value with a precision *p* and scale *s*. (The maximum precision is driver-defined.) (1 <= *p* <= 16; *s* <= *p*). [4]
SQL_NUMERIC	NUMERIC(p,s)	Signed, exact, numeric value with a precision *p* and scale *s*. (1 <= *p* <= 16; *s* <= *p*). [4]
SQL_SMALLINT	SMALLINT	Exact numeric value with precision 5 and scale 0. (signed: -32,768 <=n <= 32,767, unsigned: 0 <= n <= 65,535) solidDB supports only signed, not unsigned, SMALLINT. [5]
SQL_INTEGER	INTEGER	Exact numeric value with precision 10 and scale 0. (signed: -2^{31} <= n <= 2^{31} -1, unsigned: 0 <= n <= 2^{32} -1) solidDB supports only signed, not unsigned, INTEGER. [5]
SQL_REAL	REAL	Signed, approximate, numeric value with a binary precision 24 (zero or absolute value 10^{-38} to 1038).
SQL_FLOAT	FLOAT(p)	Signed, approximate, numeric value with a binary precision of at least *p*. (The maximum precision is driver defined.) [6]
SQL_DOUBLE	DOUBLE PRECISION	Signed, approximate, numeric value with a binary precision 53 (zero or absolute value 10-308 to 10 308).
SQL_BIT	BIT	Single bit binary data. NOTE: solidDB does not support BIT/SQL_BIT. [7]

Table 152. Common SQL Data Type Names, Ranges, and Limits (continued)

SQL Type Identifier [1]	Typical SQL Data Type [2]	Typical Type Description
SQL_TINYINT	TINYINT	Exact numeric value with precision 3 and scale 0 (signed:-128 <= *n* <= 127 unsigned: 0 <= *n* <= 255) solidDB supports only signed, not unsigned, TINYINT. [5].
SQL_BIGINT	BIGINT	Exact numeric value with precision 19 (if signed) or 20 (if unsigned) and scale 0 (signed: -2^{63} <= *n* <= 2^{63} - 1, unsigned: 0 <= *n* <= 2^{64} - 1) solidDB supports only signed, not unsigned, BIGINT. [3], [5].
SQL_BINARY	BINARY(n)	Binary data of fixed length *n*. [3]
SQL_VARBINARY	VARBINARY(n)	Variable length binary data of maximum length *n*. The maximum is set by the user. [3]
SQL_LONGVARBINARY	LONG VARBINARY	Variable length binary data. Maximum length is data source-dependent. [3]
SQL_TYPE_DATE [8]	DATE	Year, month, and day fields, conforming to the rules of the Gregorian calendar. (See "Constraints of the gregorian calendar" on page 222.)
SQL_TYPE_TIME [8]	TIME(p)	Hour, minute, and second fields. Valid values for hours are 00 to 23. Valid values for minutes are 00 to 59. Valid values for seconds are 00 to 61 (60 and 61 are to handle "leap seconds" (see http://tycho.usno.navy.mil/leapsec.html). Precision *p* indicates the precision of the seconds field.
SQL_TYPE_TIMESTAMP [8]	TIMESTAMP(p)	Year, month, day, hour, minute, and send fields, with valid values as defined for the DATE and Time data types.

Note:

[1] This is the value returned in the DATA_TYPE column by a call to SQLGetTypeInfo.

[2] This is the value returned in the NAME and CREATE PARAMS column by a call to SQLGetTypeInfo. The NAME column returns the designation - for example, CHAR - while the CREATE PARAMS column returns a comma-separated list of creation parameters such as precision, scale, and length.

[3] This data type has no corresponding data type in SQL-92.

[4] SQL_DECIMAL and SQL_NUMERIC data types differ only in their precision. The precision of a DECIMAL(p,s) is an implementation-defined decimal precision that is no less than p, while the precision of a NUMERIC(p,s) is exactly equal to p.

[5] An application uses SQLGetTypeInfo or SQLColAttribute to determine if a particular data type or a particular column in a result set is unsigned.

[6] Depending on the implementation, the precision of SQL_FLOAT can be either 24 or 53: if it is 24, the SQL_FLOAT data type is the same as SQL_REAL; if it is 53, the SQL_FLOAT data type is the same as SQL_DOUBLE.

[7] The SQL_BIT data type has different characteristics than the BIT type in SQL-92.

[8] This data type has no corresponding data type in SQL-92.

SQLGetTypeInfo Result Set Example

Applications call SQLGetTypeInfo result set for a list of supported data types and their characteristics for a given data source.

The example below shows the data types that SQLGetTypeInfo returns for a data source; all data types under "DATA_TYPE" are supported in this data source.

The example below is divided into 3 sections so that it fits the width of a page. In fact, it is all one example.

Table 153. Data types SQLGetTypeInfo returns (1)

TYPE_ NAME	DATA_TYPE	COLUMN_ SIZE	LITERAL_ PREFIX	LITERAL_ SUFFIX	CREATE_ PARAMS	NULLABLE
"char"	SQL_CHAR	255	""	""	"length"	SQL_TRUE
"text"	SQL_LONG VARCHAR	2147483647	""	""	<Null>	SQL_TRUE
"decimal"	SQL_DECIMAL	18 [a]	<Null>	<Null>	"precision, scale"	SQL_TRUE
"real"	SQL_REAL	7	<Null>	<Null>	<Null>	SQL_TRUE
"datetime"	SQL_TYPE_ TIMESTAMP	29 [b]	""	""	<Null>	SQL_TRUE

Table 154. Data Types SQLGetTypeInfo Returns (2)

(continued)	CASE_ SENSITIVE	SEARCHABLE	UNSIGNED_ ATTRIBUTE	FIXED_ PREC_ SCALE	AUTO_ UNIQUE_ VALUE	LOCAL_ TYPE_ NAME
SQL_CHAR	SQL_FALSE	SQL_ SEARCHABLE	<Null>	SQL_FALSE	<Null>	"char"
SQL_LONG VARCHAR	SQL_FALSE	SQL_PRED_ CHAR	<Null>	SQL_FALSE	<Null>	"text"
SQL_DECIMAL	SQL_FALSE	SQL_PRED_ BASIC	SQL_FALSE	SQL_FALSE	SQL_FALSE	"decimal"
SQL_REAL	SQL_FALSE	SQL_PRED_ BASIC	SQL_FALSE	SQL_FALSE	SQL_FALSE	"real"
SQL_TYPE_ TIMESTAMP	SQL_FALSE	SQL_ SEARCHABLE	<Null>	SQL_FALSE	<Null>	"datetime"

Table 155. Data Types SQLGetTypeInfo Returns (3)

(continued)	MINIMUM_ SCALE	MAXIMUM_ SCALE	SQL_DATA_ TYPE	SQL_ DATETIME_ SUB	NUM_ PREC_ RADIX	INTERVAL_ PRECISION
SQL_CHAR	<Null>	<Null>	SQL_CHAR	<Null>	<Null>	<Null>

Table 155. Data Types SQLGetTypeInfo Returns (3) (continued)

(continued)	MINIMUM_ SCALE	MAXIMUM_ SCALE	SQL_DATA_ TYPE	SQL_ DATETIME_ SUB	NUM_ PREC_ RADIX	INTERVAL_ PRECISION
SQL_LONG VARCHAR	<Null>	<Null>	SQL_LONG VARCHAR	<Null>	<Null>	<Null>
SQL_DECIMAL	0	16	SQL_ DECIMAL	<Null>	10	<Null>
SQL_REAL	<Null>	<Null>	SQL_REAL	<Null>	10	<Null>
SQL_TYPE_ TIMESTAMP	3	3	SQL_ DATETIME	SQL_CODE_ TIMESTAMP	<Null>	12

Explanations of Footnote Numbering in the Table Above

[a] 16 digits, 1 decimal point, and an optional sign character for negative numbers

[b] 29 characters to display yyyy-mm-dd hh:MM:ss.nnnnnnnnn

C data types

The solidDB ODBC Driver supports all C data types in keeping with the need for character SQL type conversion to and from all C types.

The C data type is specified in the following functions:

- SQLBindCol and SQLGetData functions with the TargetType argument.
- SQLBindParameter with the ValueType argument.
- SQLSetDescField to set the SQL_DESC_CONCISE_TYPE field of an ARD1 or APD2
- SQLSetDescRec with the Type argument, SubType argument (if needed), and the DescriptorHandle argument set to the handle of an ARD or APD.
  - ARD: Application Row Descriptors contain information about application variables that are bound to columns returned by an SQL statement. The information includes the addresses, lengths, and C data types of the bound variables.
  - APD: Application Parameter Descriptors contain information about application variables that are bound to the parameter markers ("?") used in an SQL statement, e.g.

    ```
    SELECT * FROM table1 WHERE id = ?
    ```

    The information in the descriptors includes the addresses, lengths, and C data types of the bound variables.

The table below contains the following three columns:

C Type Identifiers
: The first column shows the C Type Identifiers that are passed to functions like SQLBindCol to indicate the type of the variable that will be bound to the column. In the following example, SQL_C_DECIMAL is a C Type Identifier:

```
// Bind MySharedVariable to column 1 of the result set. (Column 1 is a
// DECIMAL column.) The C Type Identifier SQL_C_DECIMAL shows that the
// variable MySharedVariable is of a type equivalent to DECIMAL.
SQLBindCol(..., 1, SQL_C_DECIMAL, &MySharedVariable, ...);
```

ODBC C Data Type
The second columns shows the ODBC C Data Type that is associated with each C Type Identifier. This ODBC C data type is a "typedef" that you use to define variables in your ODBC program. This helps insulate your program from platform-specific requirements. For example, if you have a column of type SQL FLOAT and you want to bind a variable to that column, you can declare your variable to be of type SQLFLOAT in the following way:

```
SQLFLOAT MySharedVariable; // Can be bound to a column of type SQL FLOAT.
```

C Type
The third column contains an example of a C type definition that corresponds to the ODBC C Data Type "typdef". The examples in this column show the most frequently used definitions on 32-bit platforms.

Note: The data types specified in this column are not platform-independent; they are examples.

```
// A portable way to declare a variable that will be bound to a column of
// type SQL FLOAT.
SQLFLOAT MySharedSQLFLOATVariable = 0.0;
// A non-portable way to declare a variable that will be bound to a column
// of type SQL INTEGER. This declaration works properly on most 32-bit
// platforms, but may fail on 64-bit platforms.
long int MySharedSQLINTEGERVariable = 0;
// Bind MySharedSQLFLOATVariable to column 1 of the result set.
SQLBindCol(..., 1, SQL_C_DOUBLE, &MySharedSQLFLOATVariable, ...);
// Bind MySharedSQLINTEGERVariable to column 2 of the result set.
SQLBindCol(..., 2, SQL_C_SLONG, &MySharedSQLINTEGERVariable, ...);
```

The C Type Identifier and the ODBC C Type do not always have similar names. The C Type Identifier has a name based on the C language data type (e.g. "float"), while the ODBC C Typedef has a name that is based on the SQL data type. Since C-language "float" corresponds to SQL "REAL", the table lists "SQL_C_FLOAT" as the C Type Identifier that corresponds to the ODBC C Typedef "SQLREAL".

Table 156. C vs ODBC Naming Correspondence

C Type identifier	ODBC C Typedef	C Type
SQL_C_CHAR	SQLCHAR	unsigned char
SQL_C_STINYINT	SCHAR	char
SQL_C_UTINYINT [i]	UCHAR	unsigned char
SQL_C_SSHORT [h]	SQLSMALLINT	short int
SQL_C_USHORT [h] [i]	SQLUSMALLINT	unsigned short int
SQL_C_SLONG [h]	SQLINTEGER	int
SQL_C_ULONG [h] [i]	SQLUINTEGER	unsigned int

Table 156. C vs ODBC Naming Correspondence (continued)

C Type identifier	ODBC C Typedef	C Type
SQL_C_SBIGINT	SQLBIGINT	_int64 [g]
SQL_C_UBIGINT [i]	SQLUBIGINT	unsigned _int64 [g] solidDB does not support unsigned data types such as this.
SQL_C_FLOAT	SQLREAL	float
SQL_C_DOUBLE	SQLDOUBLE SQLFLOAT	double
SQL_C_NUMERIC	SQLNUMERIC	unsigned char [f]
SQL_C_DECIMAL	SQLDECIMAL	unsigned char [f]
SQL_C_BINARY	SQLCHAR *	unsigned char *

SQL_C_TYPE_DATE [c]	SQL_DATE_STRUCT	`struct tagDATE_STRUCT{`
		`SQLSMALLINT year;`
		`SQLUSMALLINT month;`
		`SQLUSMALLINT day;`
		`} DATE_STRUCT; [a]`

SQL_C_TYPE_TIME [c]	SQL_TIME_STRUCT	`struct tagTIME_STRUCT {`
		`SQLUSMALLINT hour;`
		`SQLUSMALLINT minute;[d]`
		`SQLUSMALLINT second;[e]`
		`}`

SQL_C_TYPE_TIMESTAMP [c]	SQL_TIMESTAMP_STRUCT	`struct tagTIMESTAMP_STRUCT {`
		`SQLSMALLINT year; [a]`
		`SQLUSMALLINT month; [b]`
		`SQLUSMALLINT day; [c]`
		`SQLUSMALLINT hour;`
		`SQLUSMALLINT minute; [d]`
		`SQLUSMALLINT second;[e]`
		`SQLUINTEGER fraction;`
		`}`

Table 156. C vs ODBC Naming Correspondence (continued)

C Type identifier	ODBC C Typedef	C Type
Note: [a] The values of the year, month, day, hour, minute, and second fields in the datetime C data types must conform to the constraints of the Gregorian calendar. (See "Constraints of the gregorian calendar" on page 222.) [b] The value of the fraction field is the number of nanoseconds (billionths of a second) and ranges from 0 through 999,999,999 (1 less than 1 billion). For example, the value of the fraction field for a half-second is 500,000,000, for a thousandth of a second (one millisecond) is 1,000,000, for a millionth of a second (one microsecond) is 1,000, and for a billionth of a second (one nanosecond) is 1. [c] In ODBC 2.x, the C date, time, and timestamp data types are SQL_C_DATE, SQL_C_TIME, and SQL_C_TIMESTAMP. [d] A number is stored in the val field of the SQL_NUMERIC_STRUCT structure as a scaled integer, in little endian mode (the leftmost byte being the least-significant byte). For example, the number 10.001 base 10, with a scale of 4, is scaled to an integer of 100010. Because this is 186AA in hexadecimal format, the value in SQL_NUMERIC_STRUCT would be "AA 86 01 00 00 ... 00", with the number of bytes defined by the SQL_MAX_NUMERIC_LEN #define. [e] The precision and scale fields of the SQL_C_NUMERIC data type are never used for input from an application, only for output from the driver to the application. When the driver writes a numeric value into the SQL_NUMERIC_STRUCT, it will use its own driver-specific default as the value for the precision field, and it will use the value in the SQL_DESC_SCALE field of the application descriptor (which defaults to 0) for the scale field. An application can provide its own values for precision and scale by setting the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the application descriptor. [f] The DECIMAL and NUMERIC data types take up more than one byte/character. The data types will actually be declared as arrays based on the precision required for the column. For example, a column of type SQL DECIMAL(10,4) might be declared as SQL_DECIMAL[13] to take into account the 10 digits, the sign character, the decimal point character, and the string terminator. [g] _int64 might not be supplied by some compilers. [h] _SQL_C_SHORT, SQL_C_LONG, and SQL_C_TINYINT have been replaced in ODBC by signed and unsigned types: SQL_C_SSHORT and SQL_C_USHORT, SQL_C_SLONG and SQL_C_ULONG, and SQL_C_STINYINT and SQL_C_UTINYINT. An ODBC 3.x driver that should work with ODBC 2.x applications should support SQL_C_SHORT, SQL_C_LONG, and SQL_C_TINYINT, because when they are called, the Driver Manager passes them through to the driver. [i] solidDB does not support unsigned SQL data types. You may bind an unsigned C data type to a signed SQL column, but you should not do this unless the values stored in the SQL column and the C variable are within the valid range for both data types. For example, since signed TINYINT columns hold values from -128 to +127, while unsigned SQL_C_UTINYINT variables hold values from 0 to 255, you may only store values between 0 and +127 in the column and bound variable if you want the values to be interpreted properly.		

64-Bit Integer Structures

On Microsoft C compilers, the C data type identifiers SQL_C_SBIGINT and SQL_C_UBIGINT are defined as _int64. When a non-Microsoft C compiler is used, the C type may differ. If the compiler in use is supporting 64-bit integers natively, then define the driver or application ODBCINT64 as the native 64-bit integer type. If the compiler in use does not support 64-bit integers natively, define the following structures to ensure access to these C types:

```
typedef struct{
SQLUINTEGER dwLowWord;
SQLUINTEGER dwHighWord;
} SQLUBIGINT

typedef struct {
SQLUINTEGER dwLowWord;
SQLINTEGER sdwHighWord;
} SQLBIGINT
```

Because a 64-bit integer is aligned to the 8-byte boundary, be sure to align these structures to an 8-byte boundary.

Note:

solidDB supports signed BIGINT, but not unsigned BIGINT.

Default C data types

In applications that specify SQL_C_DEFAULT in SQLBindCol, SQLGetData, or SQLBindParameter, the driver assumes that the C data type of the output or input buffer corresponds to the SQL data type of the column or parameter to which the buffer is bound.

Important: To avoid compatibility problems when using different platforms, we strongly recommend that you avoid using SQL_C_DEFAULT. Instead, specify the C type of the buffer in use.

Drivers cannot always determine the correct default C type for these reasons:

- The DBMS may have promoted an SQL data type of a column or a parameter; in this case, the driver is unable to determine the original SQL data type and consequently, cannot determine the corresponding default C data type.
- The DBMS determined whether the data type of a column or parameter is signed or unsigned; in this case, the driver is unable to determine this for a particular SQL data type and consequently, cannot determine this for the corresponding default C data type.

See “Converting data from SQL to C data types” on page 223.

SQL_C_TCHAR

The SQL_C_TCHAR type identifier is used for Unicode purposes. Use this identifier in applications that transfer character data and are compiled to use both ASCII and Unicode character sets. Note that the SQL_C_TCHAR is not a type identifier in the conventional sense; instead, it is a macro contained in the header file for Unicode conversion. SQL_C_CHAR or SQL_C_WCHAR replaces SQL_C_TCHAR depending on the setting of the UNICODE #define.

Numeric literals

To store numeric data values in character strings, you use numeric literals.

Numeric literal syntax specifies what is stored in the target during the following conversions:

- SQL data to an SQL_C_CHAR string
- C data to an SQL_CHAR or SQL_VARCHAR string

The syntax also validates what is stored in the source during the following conversions:

- numeric stored as an SQL_C_CHAR string to numeric SQL data
- numeric stored as an SQL_CHAR string to numeric C data

For more information, see “Numeric Literal Syntax” on page 201.

Conversion Rules

The following rules apply to conversions involving numeric literals. Following are terms used in this topic:

Table 157. Conversions Involving Numeric Literals

Term	**Meaning**
Store assignment	Refers to sending data into a table column in a database when calling SQLExecute and SQLExecDirect. During store assignment, "target" refers to a database column and "source" refers to data in application buffers.
Retrieval assignment	Refers to retrieving data from the database into application buffers when calling SQLFetch, SQLGetData, and SQLFetchScroll. During retrieval assignment, "target" refers to the application buffers and "source" refers to the database column.
CS	Value in the character source.
NT	Value in the numeric target.
NS	Value in the numeric source.
CT	Value in the character target.
Precision of an exact numeric literal	Number of digits that the literal contains.
Scale of an exact numeric literal	Number of digits to the right of the expressed or implied decimal point.
Precision of an approximate numeric literal	Precision of the literal's mantissa.

Rules for Character Source to Numeric Target

Following are the rules for converting from a character source (CS) to a numeric target (NT):

1. Replace CS with the value obtained by removing any leading or trailing spaces in CS. If CS is not a valid numeric-literal, SQLSTATE 22018 (Invalid character value for cast specification) is returned.
2. Replace CS with the value obtained by removing leading zeroes before the decimal point, trailing zeroes after the decimal point, or both.
3. Convert CS to NT. If the conversion results in a loss of significant digits, SQLSTATE 22003 (Numeric value out of range) is returned. If the conversion results in the loss of nonsignificant digits, SQLSTATE 01S07 (Fractional truncation) is returned.

Following are the rules for converting from a numeric source (NS) to a character target (CT):

1. Let LT be the length in characters of CT.

   For retrieval assignment, LT is equal to the length of the buffer in characters minus the number of bytes in the null-termination character for this character set.
2. Take one of the following actions depending on the type of NS.
   - If NS is an exact numeric type, then let YP equal the shortest character string that conforms to the definition of exact-numeric-literal such that the scale of YP is the same as the scale of NS, and the interpreted value of YP is the absolute value of NS.
   - If NS is an approximate numeric type, then let YP be a character string as follows:

   Case:

   a. If NS is equal to 0, then YP is the string "0".
   b. Let YSN be the shortest character string that conforms to the definition of exact-numeric-literal and whose interpreted value is the absolute value of NS. If the length of YSN is less than the (precision + 1) of the data type of NS, then let YP equal YSN.
   c. Otherwise, YP is the shortest character string that conforms to the definition of approximate-numeric-literal whose interpreted value is the absolute value of NS and whose mantissa consists of a single digit that is not '0', followed by a period and an unsigned-integer.
3. If NS is less than 0, then let Y be the result of:

   `'-' || YP`

   where `'||'` is the string concatenation operator.

   Otherwise, let Y equal YP.
4. Let LY be the length in characters of Y.

5. Take one of the following action depending on the value of LY.
   - If LY equals LT, then CT is set to Y.
   - If LY is less than LT, then CT is set to Y extended on the right by appropriate number of spaces.
   - Otherwise (LY > LT), copy the first LT characters of Y into CT.

     Case:
     – If this is a store assignment, return the error SQLSTATE 22001 (String data, right-truncated).
     – If this is retrieval assignment, return the warning SQLSTATE 01004 (String data, right-truncated). When the copy results in the loss of fractional digits (other than trailing zeros), depending on the driver definition, one of the following actions occurs:
     a. The driver truncates the string in Y to an appropriate scale (which can be zero also) and writes the result into CT.
     b. The driver rounds the string in Y to an appropriate scale (which can be zero also) and writes the result into CT.
     c. The driver neither truncates nor rounds, but just copies the first LT characters of Y into CT.

Overriding default precision and scale for numeric data types

The following table provides the override default precision and scale values for numeric data type.

Table 158. Override Default Precision and Scale Values for Numeric Data Type

Function calls to	Setting	Override
SQLBindCol or SQLSetDescField	SQL_DESC_TYPE field in an ARD is set to SQL_C_NUMERIC	SQL_DESC_SCALE field in the ARD is set to 0 and the SQL_DESC_PRECISION field is set to a driver-defined default precision. [a]
SQLBindParameter or SQLSetDescField	SQL_DESC_SCALE field in an APD is set to SQL_C_NUMERIC	SQL_DESC_SCALE field in the ARD is set to 0 and the SQL_DESC_PRECISION field is set to a driver-defined default precision. This is true for input, input/output, or output parameters. [a]
SQLGetData	Data is returned into an SQL_C_NUMERIC structure	Default SQL_DESC_SCALE and SQL_DESC_PRECISION fields are used. [b]

Explanations of Footnote Numbering in the Table Above

[a] If the defaults are not acceptable for an application, the application can call the SQLSetDescField or SQLSetDescRec to set the SQL_DESC_SCALE or SQL_DESC_PRECISION field.

[b] If the defaults are not acceptable, the application must call SQLSetDescRec or SQLSetDescField to set the fields and then call SQLGetData with a TargetType of SQL_ARD_TYPE to use the values in the descriptor fields.

Data type identifiers and descriptors

Unlike the "concise" SQL and C data types, where each identifier refers to a single data type, descriptors do not in all cases use a single value to identify data types. In some cases, descriptors use a verbose data type and a type subcode. For most data types, the verbose data type identifier matches the concise type identifier.

The exception, however, is the datetime and interval data types. For these data types:
- SQL_DESC_TYPE contains the verbose type (SQL_DATETIME)
- SQL_DESC_CONCISE_TYPE contains a concise type

For details on setting fields and a setting's effect on other fields, see the SQLSetDescField function description on the Microsoft ODBC web site.

When the SQL_DESC_TYPE or SQL_DESC_CONCISE_TYPE field is set for some data types, the following fields are set to default values appropriate for the data type:
- SQL_DESC_DATETIME_INTERVAL_PRECISION
- SQL_DESC_LENGTH
- SQL_DESC_PRECISION
- SQL_DESC_SCALE

For more information, see the SQL_DESC_TYPE field under SQLSetDescField function description on the Microsoft ODBC web site.

Note: If the default values set are not appropriate, you can explicitly set the descriptor field in the application by calling SQLSetDescField.

The following table lists for each SQL and C type identifier, the concise type identifier, verbose identifier, and type subcode for each datetime.

For datetime data types, the SQL_DESC_TYPE have the same manifest constants for both SQL data types (in implementation descriptors) and for C data types (in application descriptors):

Table 159. Concise Type Identifier, Verbose Identifier, and Type Subcode for Each Datetime

Concise SQL Type	Concise C Type	Verbose Type	DATETIME_INTERVAL_CODE (also called "type subcode")
SQL_TYPE_DATE	SQL_C_TYPE_DATE	SQL_DATETIME	SQL_CODE_DATE
SQL_TYPE_TIME	SQL_C_TYPE_TIME	SQL_DATETIME	SQL_CODE_TIME
SQL_TYPE_TIMESTAMP	SQL_C_TYPE_TIMESTAMP	SQL_DATETIME	SQL_CODE_TIME STAMP

Pseudo-type identifiers

ODBC defines a number of pseudo-type identifiers, which depending on the situation, resolve to existing data types. Tthese identifiers do not correspond to actual data types, but are provided for your application programming convenience.

Decimal digits

Decimal digits apply to decimal and numeric data types. They refer to the maximum number of digits to the right of the decimal point, or the scale of the data.

Because the number of digits to the right of the decimal point is not fixed, the scale is undefined for approximate floating-point number columns or parameters. When datetime data contains a seconds component, the decimal digits are the number of digits to the right of the decimal point in the seconds component of the data.

Typically, the maximum scale matches the maximum precision for SQL_DECIMAL and SQL_NUMERIC data types. Some data sources, however, have their own maximum scale limit. An application can call SQLGetTypeInfo to determine the minimum and maximum scales allowed for a data type.

The following ODBC functions return parameter decimal attributes in an SQL statement data type or decimal attributes on a data source:

Table 160. ODBC Functions' Return Parameter

ODBC Function	Returns...
SQLDescribeCol	Decimal digits of the columns it describes.
SQLDescribeParam	Decimal digits of the parameters it describes.
SQLProcedureColumns	Decimal digits in a column of a procedure.
SQLColumns	Decimal digits in specified tables (such as the base table, view, or a system table).
SQLColAttribute	Decimal digits of columns at the data source.
SQLGetTypeInfo	Minimum and maximum decimal digits of an SQL data type on a data source.

Note: The SQLBindParameter sets the decimal digits for a parameter in an SQL statement.

The values returned by ODBC functions for decimal digits correspond to "scale" as defined in ODBC 2.x.

Descriptor fields describe the characteristics of a result set. They do not contain valid data values before statement execution. However, the decimal digits values returned by SQLColumns, SQLProcedureColumns, and SQLGetTypeInfo, do represent the characteristics of database objects, such as table columns and data types form the data source's catalog.

Each concise SQL data type has the following decimal digits definition as noted in the following table:

Table 161. SQL data type decimal digits

SQL Type Identifier	Decimal Digits
All character and binary types [a]	N/A
SQL_DECIMAL SQL_NUMERIC	The defined number of digits to the right of the decimal point. For example, the scale of a column defined as NUMERIC(10,3) is 3. (In some implementations, this can be a negative number to support storage of very large numbers without using exponential notation; for example, "12000" could be stored as "12" with a scale of -3. However, solidDB does not support negative scale.)
All exact numeric types other than SQL_DECIMAL and SQL_NUMERIC [a]	0
All approximate data types [a]	N/A
Note: [a] SQLBindParameter's DecimalDigits argument is ignored for this data type.	

For decimal digits, the values returned do not correspond to the values in any one descriptor field. The values returned (for example, in SQLColAttribute) for the decimal digits can come from either the SQL_DESC_SCALE or the SQL_DESC_PRECISION field, depending on the data type, as shown in the following table:

Table 162. Descriptor field corresponding to decimal digits

SQL Type Identifier	Descriptor field corresponding to decimal digits
All character and binary types	N/A
All exact numeric types	SCALE
All approximate numeric types	N/A
All datetime types	PRECISION

Transfer octet length

When data is transferred to its default C data type, an application receives a maximum number of bytes. This maximum is known as the transfer octet length of a column.

For character data, space for the null-termination character is not included in the transfer octet length. Note that the transfer octet length in bytes can differ from the number of bytes needed to store the data on the data source.

The following ODBC functions return parameter decimal attributes in an SQL statement data type or decimal attributes on a data source:

Table 163. ODBC Functions' Return parameter Decimal Attributes

ODBC Function	Returns
SQLColumns	Transfer octet length of a column in specified tables (such as the base table, view, or a system table).
SQLColAttribute	Transfer octet length of columns at the data source.
SQLProcedureColumns	Transfer octet length of a column in a procedure.

The values returned by ODBC functions for the transfer octet length may not correspond to the values returned in SQL_DESC_LENGTH. For all character and binary types, the values come from a descriptor field's SQL_DESC_OCTET_LENGTH. For other data types, there is no descriptor field that stores this information.

Descriptor fields describe the characteristics of a result set. They do not contain valid data values before statement execution. In its result set, SQLColAttribute returns the transfer octet length of columns at the data source; these values may not match the values in the SQL_DESC_OCTET_LENGTH descriptor fields. For more information on descriptor fields, see SQLSetDescField function description on the Microsoft ODBC Web site.

Each concise SQL data type has the following transfer octet length definition as noted in the table below.

Table 164. Transfer Octet Lengths

SQL Type Identifier	Transfer Octet Length
All character and binary types [a]	The defined or the maximum (for variable type) length of the column in bytes. This value matches the one in the SQL_DESC_OCTET_LENGTH descriptor field.
SQL_DECIMAL SQL_NUMERIC	The number of bytes required to hold the character representation of this data if the character set is ASCII, and twice this number if the character set is UNICODE. The character representation is the maximum number of digits plus two; the data is returned as a character string, where the characters are needed for digits, a sign, and a decimal point. For example, the transfer length of a column defined as NUMERIC(10,3) is 12 because there are 10 bytes for the digits, 1 byte for the sign, and 1 byte for the decimal point.
SQL_TINYINT	1
SQL_SMALLINT	2
SQL_INTEGER	4

Table 164. Transfer Octet Lengths (continued)

SQL Type Identifier	Transfer Octet Length
SQL_BIGINT	The number of bytes required to hold the character representation of this data if the character set is ASCII, and twice this number if the character set is UNICODE. This data type is returned as a character string by default. The character representation consists of 20 characters for 19 digits and a sign (if signed), or 20 digits (if unsigned). The length is 20. solidDB supports only signed, not unsigned, BIGINT.
SQL_REAL	4
SQL_FLOAT	8
SQL_DOUBLE	8
All binary types [a]	The number of bytes required to store the defined (for fixed types) or maximum (for variable types) number of characters.
SQL_TYPE_DATE SQL_TYPE_TIME	6 (size of the structures SQL_DATE_STRUCT or SQL_TIME_STRUCT).
SQL_TYPE_TIMESTAMP	16 (size of the structure SQL_TIMESTAMP_STRUCT).

Explanations of Footnote Numbering in the Table Above

[a] SQL_NO_TOTAL is returned when the driver cannot determine the column or parameter length for variable types.

Constraints of the gregorian calendar

The following table contains the Gregorian calendar constraints for date and datetime data types.

Table 165. Constraints of the Gregorian Calendar

Value	Requirement
month field	Must be between 1 and 12, inclusive.
day field	Range must be from 1 through the number of days in the month, which is determined from the values of the year and months fields and can be 28, 29, 30, or 31. A leap year can also affect the number of days in the month.
hour field	Must be between 0 and 23, inclusive.
minute field	Must be between 0 and 59, inclusive.

Table 165. Constraints of the Gregorian Calendar (continued)

Value	Requirement
trailing seconds field	Must be between 0 and 61.9(n), inclusive, where n specifies the number of digits at the place of "9" and the value of n is the fractional seconds precision. The range of seconds permits a maximum of two leap seconds to maintain synchronization of sidereal time.

Converting data from SQL to C data types

This section provides information about converting data from SQL to C data types.

When an application calls SQLFetch, SQLFetchScroll, or SQLGetData, the driver retrieves the data from the data source. If necessary, it converts the data from the data type in which the driver retrieved it to the data type specified by the TargetType argument in SQLBindCol or SQLGetData. Finally, it stores the data in the location pointed to by the TargetValuePtr argument in SQLBindCol or SQLGetData (and the SQL_DESC_DATA_PTR field of the ARD).

The following table shows the supported conversions from ODBC SQL data types to ODBC C data types. A solid circle indicates the default conversion for an SQL data type (the C data type to which the data will be converted when the value of TargetType is SQL_C_DEFAULT). A hollow circle indicates a supported conversion.

For an ODBC 3.x application working with an ODBC 2.x driver, conversion from driver-specific data types might not be supported.

The format of the converted data is not affected by the Microsoft Windows country setting.

solidDB supports only signed, not unsigned, integer data types (SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT). You may bind an unsigned C variable to a signed SQL column, but you must make sure that the values you store fit within the range supported by both data types.

solidDB does not support the BIT/SQL_BIT data type for SQL columns. However, you may bind a numeric SQL column to a BIT data type in your C application. For example, you may use a TINYINT column in your database and bind that column to a C variable of type SQL_C_BIT. The solidDB ODBC driver will try to convert numeric types in the database to BIT data types for the C variables. The numeric data values must be 1 or 0 or NULL; other values cause a data conversion error. The table below does not discuss BIT/SQL_BIT data types.

CAUTION:
Although the table shows a wide range of ODBC conversions, including conversions involving unsigned data types, solidDB supports only signed integer data types (e.g. TINYINT, SMALLINT, INTEGER, and BIGINT).

Table 166. C Data Type — SQL_C_datatype where Datatype Is:

SQL Data Type	CHAR	WCHAR	STINYINT	UTINYINT	TINYINT	SSHORT	USHORT	SHORT	SLONG	ULONG	LONG	SBIGINT	UBIGINT	FLOAT	DOUBLE	NUMERIC	BINARY	DATE	TIME	TIMESTAMP
SQL_CHAR	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_VARCHAR	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_LONGVARCHAR	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WCHAR	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WVARCHAR	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WLONGVARCHAR	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_TINYINT (signed)	o	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_TINYINT (unsigned)	o	o	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_SMALLINT (signed)	o	o	o	o	o	*	o	o	o	o	o	o	o	o	o	o	o			
SQL_SMALLINT (unsigned)	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o	o	o			
SQL_INTEGER (signed)	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o			
SQL_INTEGER (unsigned)	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o			
SQL_BIGINT (signed)	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o				
SQL_BIGINT (unsigned)	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o			
SQL_REAL	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o			
SQL_FLOAT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o			
SQL_DOUBLE	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o			
SQL_DECIMAL	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_NUMERIC	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_BINARY	o	o															*			
SQL_VARBINARY	o	o															*			
SQL_LONGVARBINARY	o	o															*			

Table 166. C Data Type — SQL_C_datatype where Datatype Is: (continued)

SQL Data Type	CHAR	WCHAR	STINYINT	UTINYINT	TINYINT	SSHORT	USHORT	SHORT	SLONG	ULONG	LONG	SBIGINT	UBIGINT	FLOAT	DOUBLE	NUMERIC	BINARY	DATE	TIME	TIMESTAMP
SQL_TYPE_DATE	o	o															o	*		o
SQL_TYPE_TIME	o	o															o		*	o
SQL_TYPE_TIMESTAMP	o	o															o	o	o	*

* These datatypes have the word "TYPE" in the datatype name. For example, SQL_C_TYPE_DATE, SQL_C_TYPE_TIME, and SQL_C_TYPE_TIMESTAMP.

Legend:

* Default Conversion

o Supported Conversion

Data conversion tables from SQL to C

The tables in the following sections describe how the driver or data source converts data retrieved from the data source; drivers are required to support conversions to all ODBC C data types from the ODBC SQL data types that they support.

Conversion Table Description (SQL to C)

The following columns are included in the tables:

- For a given ODBC SQL data type, the first column of the table lists the legal input values of the TargetType argument in SQLBindCol and SQLGetData.
- The second column lists the outcomes of a test, often using the BufferLength argument specified in SQLBindCol or SQLGetData, which the driver performs to determine if it can convert the data.
- For each outcome, the third and fourth columns list the values placed in the buffers specified by the TargetValuePtr and StrLen_or_IndPtr arguments specified in SQLBindCol or SQLGetData after the driver has attempted to convert the data. (The StrLen_or_IndPtr argument corresponds to the SQL_DESC_OCTET_LENGTH_PTR field of the ARD.)
- The last column lists the SQLSTATE returned for each outcome by SQLFetch, SQLFetchScroll, or SQLGetData.

If the TargetType argument in SQLBindCol or SQLGetData contains a value for an ODBC C data type not shown in the table for a given ODBC SQL data type, SQLFetch, SQLFetchScroll, or SQLGetData returns SQLSTATE 07006 (Restricted data type attribute violation). If the TargetType argument contains a value that specifies a conversion from a driver-specific SQL data type to an ODBC C data type and this conversion is not supported by the driver, SQLFetch, SQLFetchScroll, or SQLGetData returns SQLSTATE HYC00 (Optional feature not implemented).

Although it is not shown in the tables, the driver returns SQL_NULL_DATA in the buffer specified by the StrLen_or_IndPtr argument when the SQL data value is NULL. The length specified by StrLen_or_IndPtr does not include the null-termination byte. If TargetValuePtr is a null pointer, SQLGetData returns SQLSTATE HY009 (Invalid use of null pointer); in SQLBindCol, this unbinds the columns.

The following terms and conventions are used in the tables:
- Byte length of data is the number of bytes of C data available to return in * TargetValuePtr, whether or not the data will be truncated before it is returned to the application. For string data, this does not include the space for the null-termination character.
- Character byte length is the total number of bytes needed to display the data in character format.
- Words in italics represent function arguments or elements of the SQL grammar. See Appendix D, "Minimum SQL grammar requirements for ODBC," on page 197 for the syntax of grammar elements.

SQL to C: Character

The character ODBC SQL data types are:

```
SQL_CHAR
SQL_VARCHAR
SQL_LONGVARCHAR
SQL_WCHAR
SQL_WVARCHAR
SQL_WLONGVARCHAR
```

The following table shows the ODBC C data types to which character SQL data can be converted. For an explanation of the columns and terms in the table, see "Conversion Table Description (SQL to C)" on page 225.

Table 167. Character SQL Data to ODBC C Data Types

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	Byte length of data < BufferLength Byte length of data >= BufferLength	Data Truncated data	Length of data in bytes Length of data in bytes	N/A 01004
SQL_C_WCHAR	Character length of data < BufferLength (Character length of data) >= BufferLength	Data Truncated data	Length of data in characters Length of data in characters	N/A 01004

Table 167. Character SQL Data to ODBC C Data Types (continued)

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
EXACT NUMERIC TYPES [h] SQL_C_STINYINT SQL_C_UTINYINT SQL_C_TINYINT SQL_C_SSHORT SQL_C_USHORT SQL_C_SHORT SQL_C_SLONG SQL_C_ULONG SQL_C_LONG SQL_C_SBIGINT SQL_C_UBIGINT SQL_C_NUMERIC	Data converted without truncation [b] Data converted with truncation of fractional digits [a] Conversion of data would result in loss of whole (as opposed to fractional) digits [b] Data is not a numeric-literal [b]	Data Truncated data Undefined Undefined	Number of bytes of the C data type Number of bytes of the C data type Undefined Undefined	N/A 01S07 22003 22018
APPROXIMATE NUMERIC TYPES [h] SQL_C_FLOAT SQL_C_DOUBLE	Data is within the range of the data type to which the number is being converted [a] Data is outside the range of the data type to which the number is being converted [a] Data is not a numeric-literal [b]	Data Undefined Undefined	Size of the C data type Undefined Undefined	N/A 2003 22018
SQL_C_BINARY	Byte length of data <= BufferLength Byte length of data > BufferLength	Data Truncated data	Length of data Length of data	N/A 01004

Table 167. Character SQL Data to ODBC C Data Types (continued)

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_TYPE_DATE	Data value is a valid date-value [a] Data value is a valid timestamp-value; time portion is zero [a] Data value is a valid timestamp-value; time portion is nonzero [a], [c], Data value is not a valid date-value or timestamp_value [a]	Data Data Truncated data Undefined	6 [b] 6 [b] 6 [b] Undefined	N/A N/A 01S07 22018
SQL_C_TYPE_TIME	Data value is a valid time-value and the fractional seconds value is 0 [a] Data value is a valid timestamp-value or a valid time_value; fractional seconds portion is zero [a],[d] Data value is a valid timestamp-value ; fractional seconds portion is nonzero [a], [d], [e] Data value is not a valid timestamp-value or time_value [a]	Data Data Truncated data Undefined	6 [b] 6 [b] 6 [b] Undefined	N/A N/A 01S07 22018
SQL_C_TYPE_TIMESTAMP	Data value is a valid timestamp-value or a valid time_value; fractional seconds portion not truncated [a], [d] Data value is a valid timestamp-value or a valid time_value; fractional seconds portion truncated [a] Data value is a valid date-value [a] Data value is a valid time_value [a] Data value is not a valid date_value, time_value, or timestamp_value [a]	Data Truncated data Data [f] Data [g] Undefined	16 [b] 16 [b] 16 [b] 16 [b] Undefined	N/A 01S07 N/A N/A 22018

Table 167. Character SQL Data to ODBC C Data Types (continued)

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
Note: [a] The value of BufferLength is ignored for this conversion. The driver assumes that the size of *TargetValuePtr is the size of the C data type. [b] This is the size of the corresponding C data type. [c] The time portion of the timestamp-value is truncated. [d] The date portion of the timestamp-value is ignored. [e] The fractional seconds portion of the timestamp is truncated. [f] The time fields of the timestamp structure are set to zero. [g] The date fields of the timestamp structure are set to the current date. [h] The exact numeric types include NUMERIC/DECIMAL as well as integer. These data types store the exact value that you specify, as long as it is within the precision of the data type. The approximate data types include FLOAT/REAL, which store only approximately the value that you specify (in some cases, the least significant digit may be slightly different from what you specified).				

When character SQL data is converted to numeric, date, time, or timestamp C data, leading and trailing spaces are ignored.

SQL to C: Numeric

```
SQL_DECIMAL     SQL_BIGINT
SQL_NUMERIC     SQL_REAL
SQL_TINYINT     SQL_FLOAT
SQL_SMALLINT    SQL_DOUBLE
SQL_INTEGER
```

The following table shows the ODBC C data types to which numeric SQL data may be converted. For an explanation of the columns and terms in the table, see “Conversion Table Description (SQL to C)” on page 225.

Table 168. SQL Data to ODBC C Data Types

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	Character byte length < BufferLength Number of whole (as opposed to fractional) digits < BufferLength Number of whole (as opposed to fractional) digits ≥ BufferLength	Data Truncated data Undefined	Length of data in bytes Length of data in bytes Undefined	N/A 01004 22003

Table 168. SQL Data to ODBC C Data Types (continued)

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_WCHAR	Character length < BufferLength Number of whole (as opposed to fractional) digits < BufferLength Number of whole (as opposed to fractional) digits ≥ BufferLength	Data Truncated data Undefined	Length of data in bytes Length of data in bytes Undefined	N/A 01004 22003
EXACT NUMERIC TYPES [c] SQL_C_STINYINT SQL_C_UTINYINT SQL_C_TINYINT SQL_C_SBIGINT SQL_C_UBIGINT SQL_C_SSHORT SQL_C_USHORT SQL_C_SHORT a SQL_C_SLONG SQL_C_ULONG SQL_C_LONG SQL_C_NUMERIC	Data converted without truncation [a] Data converted with truncation of fractional digits [a] Conversion of data would result in loss of whole (as opposed to fractional) digits [a]	Data Truncated data Undefined	Size of the C data type Size of the C data type Undefined	N/A 01S07 22003
APPROXIMATE NUMERIC TYPES [c] SQL_C_FLOAT SQL_C_DOUBLE	Data is within the range of the data type to which the number is being converted [a] Data is outside the range of the data type to which the number is being converted [a]	Data Undefined	Size of the C data type Undefined	N/A 22003
SQL_C_BINARY	Length of data ≤ BufferLength Length of data > BufferLength	Data Undefined	Length of data Undefined	N/A 22003

Table 168. SQL Data to ODBC C Data Types (continued)

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
Note: [a] The value of BufferLength is ignored for this conversion. The driver assumes that the size of *TargetValuePtr is the size of the C data type. [b] This is the size of the corresponding C data type. [c] The exact numeric types include NUMERIC/DECIMAL as well as integer. These data types store the exact value that you specify, as long as it is within the precision of the data type. The approximate data types include FLOAT/REAL, which store only approximately the value that you specify (in some cases, the least significant digit may be slightly different from what you specified).				

SQL to C: Binary

The binary ODBC SQL data types are:

```
SQL_BINARY
SQL_VARBINARY
SQL_LONGVARBINARY
```

The following table shows the ODBC C data types to which binary SQL data may be converted. For an explanation of the columns and terms in the table, see “Conversion Table Description (SQL to C)” on page 225.

Table 169. Binary SQL Data to ODBC C Data Types

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	(Byte length of data) * 2 < BufferLength (Byte length of data) * 2 >= BufferLength	Data Truncated data	Length of data in bytes Length of data in bytes	N/A 01004
SQL_C_WCHAR	(Character length of data) * 2 < BufferLength (Character length of data) * 2 >= BufferLength	Data Truncated data	Length of data in bytes Length of data in bytes	N/A 01004
SQL_C_BINARY	Byte length of data <= BufferLength Byte Length of data > BufferLength	Data Truncated data	Length of data in bytes Length of data in bytes	N/A 01004

When binary SQL data is converted to character C data, each byte (8 bits) of source data is represented as two ASCII characters. These characters are the ASCII character representation of the number in its hexadecimal form. For example, a binary 00000001 is converted to "01"and a binary 11111111 is converted to "FF".

T he driver always converts individual bytes to pairs of hexadecimal digits and terminates the character string with a null byte. Because of this, if BufferLength is even and is less than the length of the converted data, the last byte of the

*TargetValuePtr buffer is not used. (The converted data requires an even number of bytes, the next-to-last byte is a null byte, and the last byte cannot be used.)

Application developers are discouraged from binding binary SQL data to a character C data type. This conversion is usually inefficient and slow.

SQL to C: Date

The date ODBC SQL data type is:

`SQL_DATE`

The following table shows the ODBC C data types to which date SQL data may be converted. For an explanation of the columns and terms in the table, see “Conversion Table Description (SQL to C)” on page 225.

Table 170. Date SQL Data to ODBC C Data Types

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	BufferLength > Character byte length 11<= BufferLength <= Character byte length BufferLength < 11	Data Truncated data Undefined	10 Length of data in bytes Undefined	N/A 01004 22003
SQL_C_WCHAR	BufferLength > Character length 11<= BufferLength <= Character length BufferLength < 11	Data Truncated data Undefined	10 Length of data in bytes Undefined	N/A 01004 22003
SQL_C_BINARY	Byte length of data <= BufferLength Byte length of data > BufferLength	Data Undefined	Length of data in bytes Undefined	N/A 22003
SQL_C_DATE	None [a]	Data	6 [c]	N/A
SQL_C_TIMESTAMP	None [a]	Data [b]	16 [c]	N/A

Note:

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

[b] The time fields of the timestamp structure are set to zero.

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

When date SQL data is converted to character C data, the resulting string is in the "yyyy-mm-dd"format. This format is not affected by the Microsoft Windows country setting.

SQL to C: Time

The time ODBC SQL data type is:

```
SQL_TIME
```

The following table shows the ODBC C data types to which time SQL data may be converted. For an explanation of the columns and terms in the table, see "Conversion Table Description (SQL to C)" on page 225.

Table 171. Time SQL Data to ODBC C Data Types

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	BufferLength > Character byte length 9 <= BufferLength <= Character byte length BufferLength < 9	Data Truncated data [a] Undefined	Length of data in bytes Length of data in bytes Undefined	N/A 01004 22003
SQL_C_WCHAR	BufferLength > Character byte length 9 <= BufferLength <= Character byte length BufferLength < 9	Data Truncated data [a] Undefined	Length of data in characters Length of data in characters Undefined	N/A 01004 22003
SQL_C_BINARY	Byte length of data <= BufferLength Byte length of data > BufferLength	Data Undefined	Length of data in bytes Undefined	N/A 22003
SQL_C_DATE	None [a]	Data	6 [c]	N/A
SQL_C_TIMESTAMP	None [a]	Data [b]	16 [c]	N/A

Note:

[a]: The fractional seconds of the time are truncated.

[b]: The value of BufferLength is ignored for this conversion. The driver assumes that the size of *TargetValuePtr is the size of the C data type.

[c]: The date fields of the timestamp structure are set to the current date and the fractional seconds field of the timestamp structure is set to zero.

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

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

SQL to C: Timestamp

The timestamp ODBC SQL data type is:

```
SQL_TIMESTAMP
```

The following table shows the ODBC C data types to which timestamp SQL data may be converted. For an explanation of the columns and terms in the table, see “Conversion Table Description (SQL to C)” on page 225.

Table 172. Timestamp SQL Data to ODBC C Data Types

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	BufferLength > Character byte length 20 <= BufferLength <= Character byte length BufferLength < 20	Data Truncated data [b] Undefined	Length of data in bytes Length of data in bytes Undefined	N/A 01004 22003
SQL_C_WCHAR	BufferLength > Character byte length 20 <= BufferLength <= Character byte length BufferLength < 20	Data Truncated data [b] Undefined	Length of data in characters Length of data in characters Undefined	N/A 01004 22003
SQL_C_BINARY	Byte length of data <= BufferLength Byte length of data > BufferLength	Data Undefined	Length of data in bytes Undefined	N/A 22003
SQL_C_TYPE_DATE	Time portion of timestamp is zero [a] Time portion of timestamp is non-zero [a]	Data Truncated data [c]	6 [f] 6 [f]	N/A 01S07
SQL_C_TYPE_TIME	Fractional seconds portion of timestamp is zero [a] Fractional seconds portion of timestamp is non-zero [a]	Data [d] Truncated data [d], [e]	6 [f] 6 [f]	N/A 01S07
SQL_C_TYPE_TIMESTAMP	Fractional seconds portion of timestamp is not truncated [a] Fractional seconds portion of timestamp is truncated [a]	Data [e] Truncated data [e]	6 [f] 6 [f]	N/A 01S07

Table 172. Timestamp SQL Data to ODBC C Data Types (continued)

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
Note: [a] The value of BufferLength is ignored for this conversion. The driver assumes that the size of *TargetValuePtr 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 truncated. [d] The date portion of the timestamp is ignored. [e] The fractional seconds portion of the timestamp is truncated. [f] This is the size of the corresponding C data type.				

When timestamp SQL data is converted to character C data, the resulting string is in the "yyyy-mm-dd hh:mm:ss [.f ...]"format, where up to nine digits may be used for fractional seconds. The format is not affected by the Microsoft Windows country setting. (Except for the decimal point and fractional seconds, the entire format must be used, regardless of the precision of the timestamp SQL data type.)

SQL to C data conversion examples

The following examples illustrate how the driver converts SQL data to C data.

Table 173. SQL to C Data Conversion Examples

SQL Type Identifier	SQL Data Values	C Type Identifier	Buffer Length	*TargetValuePtr	SQLSTATES
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0 [a]	N/A
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0 [a]	N/A
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	N/A
SQL_DECIMAL	1234.56	SQL_C_SSHORT	ignored	1234	01S07
SQL_DECIMAL	1234.56	SQL_C_STINYINT	ignored	----	22003
SQL_DOUBLE	1.2345678	SQL_C_DOUBLE	ignored	1.2345678	N/A
SQL_DOUBLE	1.2345678	SQL_C_FLOAT	ignored	1.234567	N/A
SQL_DOUBLE	1.2345678	SQL_C_STINYINT	ignored	1	N/A

Table 173. SQL to C Data Conversion Examples (continued)

SQL Type Identifier	SQL Data Values	C Type Identifier	Buffer Length	*TargetValuePtr	SQLSTATES
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0 [a]	N/A
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	10	-----	22003
SQL_TYPE_DATE	1992-12-31	SQL_C_ TIMESTAMP	ignored	1992,12,31, 0,0,0,0 [b]	N/A
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 [a]	N/A
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 [a]	01004
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	----	22003

[a] "\0"represents a null-termination byte. The driver always null-terminates SQL_C_CHAR data.

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

Converting data from C to SQL data types

This section provides information about converting data from C to SQL data types.

When an application calls SQLExecute or SQLExecDirect, the driver retrieves the data for any parameters bound with SQLBindParameter from storage locations in the application. For data-at-execution parameters, the application sends the parameter data with SQLPutData. If necessary, the driver converts the data from the data type specified by the ValueType argument in SQLBindParameter to the data type specified by the ParameterType argument in SQLBindParameter. Finally, the driver sends the data to the data source.

The following table shows the supported conversions from ODBC C data types to ODBC SQL data types. A solid circle indicates the default conversion for an SQL data type (the C data type from which the data will be converted when the value of ValueType or the SQL_DESC_CONCISE_TYPE descriptor field is SQL_C_DEFAULT). A hollow circle indicates a supported conversion.

The format of the converted data is not affected by the Microsoft Windows country setting.

solidDB supports only signed, not unsigned, integer data types (SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT). You may bind an unsigned C variable to a signed SQL column, but you must make sure that the values you store fit within the range supported by both data types.

solidDB does not support the BIT/SQL_BIT data type for SQL columns. However, you may bind a numeric SQL column to a BIT data type in your C application. For example, you may use a TINYINT column in your database and bind that column to a C variable of type SQL_C_BIT. The solidDB ODBC driver will try to convert numeric types in the database to BIT data types for the C variables. The numeric data values must be 1 or 0 or NULL; other values cause a data conversion error. The table below does not discuss BIT/SQL_BIT data types.

CAUTION:
Although the table above shows a wide range of ODBC conversions, including conversions involving unsigned data types, solidDB supports only signed integer data types (e.g. TINYINT, SMALLINT, INT, and BIGINT).

Table 174. SQL Data Type — SQL_datatype where Datatype Is:

C Data Type	CHAR	VARCHAR	LONGVARCHAR	WCHAR	WVARCHAR	WLONGVARCHAR	DECIMAL	NUMERIC	TINYINT (signed)	TINYINT (unsigned)	SMALLINT (signed)	SMALLINT (unsigned)	INTEGER (signed)	INTEGER (unsigned)	BIGINT (signed)	BIGINT (unsigned)	REAL	FLOAT	DOUBLE	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
SQL_C_CHAR	*	*	*	o	o	o	*	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WCHAR	o	o	o	*	*	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_C_NUMERIC	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_STINYINT	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o	o	o						
SQL_C_UTINYINT	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o	o						
SQL_C_TINYINT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_SSHORT	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o						
SQL_C_USHORT	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o						
SQL_C_SHORT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_SLONG	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o						
SQL_C_ULONG	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o						
SQL_C_LONG	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_SBIGINT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	*	o	o	o						
SQL_C_UBIGINT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o						

Table 174. SQL Data Type — SQL_datatype where Datatype Is: (continued)

C Data Type	CHAR	VARCHAR	LONGVARCHAR	WCHAR	WVARCHAR	WLONGVARCHAR	DECIMAL	NUMERIC	TINYINT(signed)	TINYINT(unsigned)	SMALLINT(signed)	SMALLINT(unsigned)	INTEGER(signed)	INTEGER(unsigned)	BIGINT(signed)	BIGINT(unsigned)	REAL	FLOAT	DOUBLE	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
SQL_C_FLOAT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o						
SQL_C_DOUBLE	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	*						
SQL_C_BINARY	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	*	*	o	o	o
SQL_C_DATE	o	o	o	o	o	o																	*		o
SQL_C_TIME	o	o	o	o	o	o																		*	o
SQL_C_TIMESTAMP	o	o	o	o	o	o																	o	o	*

Legend:

* Default conversion,

o Supported conversion

Data conversion tables from C to SQL

The tables in the following sections describe how the driver or data source converts data sent to the data source; drivers are required to support conversions from all ODBC C data types to the ODBC SQL data types that they support.

Conversion Table Description (C to SQL)

The following columns are included in the tables:

- For a given ODBC C data type, the first column of the table lists the legal input values of the ParameterType argument in SQLBindParameter.
- The second column lists the outcomes of a test that the driver performs to determine if it can convert the data.
- The third column lists the SQLSTATE returned for each outcome by SQLExecDirect, SQLExecute, or SQLPutData. Data is sent to the data source only if SQL_SUCCESS is returned.

If the ParameterType argument in SQLBindParameter contains a value for an ODBC SQL data type that is not shown in the table for a given C data type, SQLBindParameter returns SQLSTATE 07006 (Restricted data type attribute violation). If the ParameterType argument contains a driver-specific value and the driver does not support the conversion from the specific ODBC C data type to that driver-specific SQL data type, SQLBindParameter returns SQLSTATE HYC00 (Optional feature not implemented).

If the ParameterValuePtr and StrLen_or_IndPtr arguments specified in SQLBindParameter are both null pointers, that function returns SQLSTATE HY009 (Invalid use of null pointer). Although it is not shown in the tables, an application sets the value pointed to by the StrLen_or_IndPtr argument of SQLBindParameter or the value of the StrLen_or_IndPtr argument to SQL_NULL_DATA to specify a NULL SQL data value. (The StrLen_or_IndPtr argument corresponds to the SQL_DESC_OCTET_LENGTH_PTR field of the APD.) The application sets these values to SQL_NTS to specify that the value in *ParameterValuePtr in SQLBindParameter or *DataPtr in SQLPutData (pointed to by the SQL_DESC_DATA_PTR field of the APD) is a null-terminated string.

The following terms are used in the tables:
- *Byte length of data* is the number of bytes of SQL data available to send to the data source, regardless of whether the data will be truncated before it is sent to the data source. For string data, this does not include the null-termination character.
- *Column byte length* is the number of bytes required to store the data at the data source.
- *Character byte length* is the maximum number of bytes needed to display data in character form.
- *Number of digits* is the number of characters used to represent a number, including the minus sign, decimal point, and exponent (if needed).
- Words in italics represent elements of the ODBC SQL grammar. For the syntax of grammar elements, see Appendix D, “Minimum SQL grammar requirements for ODBC,” on page 197.

C to SQL: Character

The character ODBC C data type is:

SQL_C_CHAR SQL_C_WCHAR

The following table shows the ODBC SQL data types to which C character data may be converted. For an explanation of the columns and terms in the table, see “Conversion Table Description (C to SQL)” on page 238.

Note: The length of the Unicode data type must be an even number when character C data is converted to Unicode SQL data.

Table 175. C Character Data to ODBC SQL Data Types

SQL Type Identifier	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Byte length of data <= Column length Byte length of data > Column length	N/A 22001

Table 175. C Character Data to ODBC SQL Data Types (continued)

SQL Type Identifier	Test	SQLSTATE
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Character length of data <= Column length Character length of data > Column length	N/A 22001
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT	Data converted without truncation Data converted with truncation of fractional digits [e] Conversion of data would result in loss of whole (as opposed to fractional) digits [e] Data value is not a *numeric-literal*	N/A 22001 22001 22018
SQL_REAL SQL_FLOAT SQL_DOUBLE	Data is within the range of the data type to which the number is being converted Data is outside the range of the data type to which the number is being converted Data value is not a *numeric-literal*	N/A 22003 22005
SQL_BIT	Data is 0 or 1 Data is greater than 0, less than 2, and not equal to 1 Data is less than 0 or greater than or equal to 2 Data is not a *numeric-literal*. Note: solidDB does not support SQL_BIT.	N/A 22001 22003 22018
SQL_BINARY SQL_VARBINARY SQL_LONG-VARBINARY	(Byte length of data) / 2 <= Column byte length (Byte length of data) / 2 > Column byte length Data value is not a hexadecimal value	N/A 22001 22018

Table 175. C Character Data to ODBC SQL Data Types (continued)

SQL Type Identifier	Test	SQLSTATE
SQL_TYPE_DATE	Data value is a valid *ODBC_date_literal* Data value is a valid *ODBC_timestamp_literal*; time portion is zero Data value is a valid *ODBC_timestamp_literal*; time portion is non-zero [a] Data value is not a valid *ODBC_date_literal* or *ODBC_timestamp_literal*	N/A N/A 22008 22018
SQL_TYPE_TIME	Data value is a *valid* *ODBC_time_literal* Data value is a valid *ODBC_timestamp_literal*; fractional seconds portion is zero [b] Data value is a valid *ODBC_timestamp_literal*; fractional seconds portion is non-zero [b] Data value is not a valid *ODBC_time_literal* or *ODBC_timestamp_literal*	N/A N/A 22008 22018
SQL_TYPE_TIMESTAMP	Data value is a valid *ODBC_timestamp_literal*; fractional seconds portion not truncated Data value is a valid *ODBC-timestamp-literal*; fractional seconds portion truncated Data value is a valid *ODBC-date-literal* [c] Data value is a valid *ODBC-time-literal* [d] Data value is not a valid *ODBC-date-literal*, *ODBC-time-literal*, or *ODBC-timestamp-literal*	N/A 22008 N/A N/A 22018

Table 175. C Character Data to ODBC SQL Data Types (continued)

SQL Type Identifier	Test	SQLSTATE
Note: [a] The time portion of the timestamp is truncated. [b] The date portion of the timestamp is ignored. [c] The time portion of the timestamp is set to zero. [d] The date portion of the timestamp is set to the current date. [e] The driver/data source effectively waits until the entire string has been received (even if the character data is sent in pieces by calls to SQLPutData) before attempting to perform the conversion.		

When character C data is converted to numeric, date, time, or timestamp SQL data, leading and trailing blanks are ignored.

When character C data is converted to binary SQL data, each two bytes of character data are converted to a single byte (8 bits) of binary data. Each two bytes of character data represent a number in hexadecimal form. For example, "01" is converted to a binary 00000001 and "FF" is converted to a binary 11111111.

The driver always converts pairs of hexadecimal digits to individual bytes and ignores the null termination byte. Because of this, if the length of the character string is odd, the last byte of the string (excluding the null termination byte, if any) is not converted.

Note: Because binding character C data to a binary SQL data type is inefficient and slow, refrain from doing this.

C to SQL: Numeric

The numeric ODBC C data types are:
- SQL_C_STINYINT
- SQL_C_SLONG
- SQL_C_UTINYINT
- SQL_C_ULONG
- SQL_C_TINYINT
- SQL_C_LONG
- SQL_C_SSHORT
- SQL_C_FLOAT
- SQL_C_USHORT
- SQL_C_DOUBLE
- SQL_C_SHORT
- SQL_C_NUMERIC
- SQL_C_SBIGINT
- SQL_C_UBIGINT

For more information about the SQL_C_TINYINT, SQL_C_SHORT, and SQL_C_LONG data types, see “C data types” on page 210. The following table shows the ODBC SQL data types to which numeric C data may be converted. For an explanation of the columns and terms in the table, see “Conversion Table Description (C to SQL)” on page 238.

Table 176. Numeric C Data to ODBC SQL Data Types

ParameterType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Number of digits <= Column byte length Number of digits > Column byte length	N/A 22001
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Number of characters <= Column character length Number of characters > Column character length	N/A 22001
SQL_DECIMAL [a] SQL_NUMERIC [a] SQL_TINYINT [a] SQL_SMALLINT [a] SQL_INTEGER [a] SQL_BIGINT [a]	Data converted without truncation or with truncated of fractional digits Data converted with truncation of whole digits	N/A 22003
SQL_REAL SQL_FLOAT SQL_DOUBLE	Data is within the range of the data type to which the number is being converted Data is outside the range of the data type to which the number is being converted	N/A 22003
Note: [a] For the "n/a" case, a driver may optionally return SQL_SUCCESS_WITH_INFO and 01S07 when there is a fractional truncation.		

The driver ignores the length or indicator value when converting data from the numeric C data types and assumes that the size of the data buffer is the size of the numeric C data type. The length or indicator value is passed in the StrLen_or_IndPtr argument in SQLPutData and in the buffer specified with the StrLen_or_IndPtr argument in SQLBindParameter. The data buffer is specified with the DataPtr argument in SQLPutData and the ParameterValuePtr argument in SQLBindParameter.

C to SQL: Bit

The bit ODBC C data type is:

```
SQL_C_BIT
```

The following table shows the ODBC SQL data types to which bit C data may be converted. For an explanation of the columns and terms in the table, see For an explanation of the columns and terms in the table, see "Conversion Table Description (C to SQL)" on page 238.

Table 177. Bit C Data to ODBC SQL Data Types

SQL Type Identifier	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	None	N/A
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE	None	N/A

The driver ignores the length or indicator value when converting data from the bit C data types and assumes that the size of the data buffer is the size of the bit C data type. The length or indicator value is passed in the StrLen_or_Ind argument in SQLPutData and in the buffer specified with the StrLen_or_IndPtr argument in SQLBindParameter. The data buffer is specified with the DataPtr argument in SQLPutData and the ParameterValuePtr argument in SQLBindParameter.

C to SQL: Binary

The binary ODBC C data type is:

SQL_C_BINARY

The following table shows the ODBC SQL data types to which binary C data may be converted. For an explanation of the columns and terms in the table, see "Conversion Table Description (C to SQL)" on page 238.

Table 178. Binary C Data to ODBC SQL Data Types

SQL Type Identifier	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Byte length of data <= Column byte length Byte length of data > Column length	N/A 22001
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Character length of data <= Column character length Character length of data > Column character length	N/A 22001
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE SQL_TYPE_DATE SQL_TYPE_TIME SQL_TYPE_TIMESTAMP	Byte length of data = SQL data length Length of data <> SQL data length	N/A 22003
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY	Length of data <= Column length Length of data > Column length	N/A 22001

C to SQL: Date

The date ODBC C data type is:

`SQL_C_DATE`

The following table shows the ODBC SQL data types to which date C data may be converted. For an explanation of the columns and terms in the table, see “Conversion Table Description (C to SQL)” on page 238.

Table 179. Date C Data to ODBC SQL Data Types

SQL Type Identifier	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column byte length >= 10 Column byte length < 10 Data value is not a valid date	N/A 22001 22008
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column character length >= 10 Column character length < 10 Data value is not a valid date	N/A 22001 22008
SQL_TYPE_DATE	Data value is a valid date Data value is not a valid date	N/A 22007
SQL_TYPE_TIMESTAMP	Data value is a valid date [a] Data value is not a valid date	N/A 22007
Note: [a] The time portion of the timestamp is set to zero.		

For information about what values are valid in an SQL_C_TYPE_DATE structure, see “C data types” on page 210.

When date C data is converted to character SQL data, the resulting character data is in the "yyyy-mm-dd "format.

The driver ignores the length or indicator value when converting data from the date C data types and assumes that the size of the data buffer is the size of the date C data type. The length or indicator value is passed in the StrLen_or_Ind argument in SQLPutData and in the buffer specified with the StrLen_or_IndPtr argument in SQLBindParameter. The data buffer is specified with the DataPtr argument in SQLPutData and the ParameterValuePtr argument in SQLBindParameter.

C to SQL: Time

The time ODBC C data type is:

SQL_C_TIME

T he following table shows the ODBC SQL data types to which time C data may be converted. For an explanation of the columns and terms in the table, see “Conversion Table Description (C to SQL)” on page 238.

Table 180. Time C Data to ODBC SQL Data Types

SQL Type Identifier	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column byte length >= 8 Column byte length < 8 Data value is not a valid time	N/A 22001 22008
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Column character length >= 8 Column character length < 8 Data value is not a valid time	N/A 22001 22008
SQL_TYPE_TIME	Data value is a valid time Data value is not a valid time	N/A 22007
SQL_TYPE_TIMESTAMP	Data value is a valid time [a] Data value is not a valid time	N/A 22007
Note: [a] The date portion of the timestamp is set to the current date and the fractional seconds portion of the timestamp is set to zero.		

For information about what values are valid in an SQL_C_TYPE_TIME structure, see "C data types" on page 210.

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

The driver ignores the length or indicator value when converting data from the time C data types and assumes that the size of the data buffer is the size of the time C data type. The length or indicator value is passed in the StrLen_or_Ind argument in SQLPutData and in the buffer specified with the StrLen_or_IndPtr argument in SQLBindParameter. The data buffer is specified with the DataPtr argument in SQLPutData and the ParameterValuePtr argument in SQLBindParameter.

C to SQL: Timestamp

The timestamp ODBC C data type is:

SQL_C_TIMESTAMP

The following table shows the ODBC SQL data types to which timestamp C data may be converted. For an explanation of the columns and terms in the table, see "Conversion Table Description (C to SQL)" on page 238.

Table 181. Timestamp C Data to ODBC SQL Data Ttypes

SQL Type Identifier	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column byte length >= Character byte length 19 <= Column byte length < Character byte length Column byte length < 19 Data value is not a valid date	N/A 22001 22001 22008
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Column character length >= Character length of data 19 <= Column character length < Character length of data Column character length < 19 Data value is not a valid timestamp	N/A 22001 22001 22008
SQL_TYPE_DATE	Time fields are zero Time fields are non-zero Data value does not contain a valid date	N/A 22008 22007
SQL_TYPE_TIME	Fractional seconds fields are zero [a] Fractional seconds fields are non-zero [a] Data value does not contain a valid time	N/A 22008 22007
SQL_TYPE_TIMESTAMP	Fractional seconds fields are not truncated Fractional seconds fields are truncated Data value is not a valid timestamp	N/A 22008 22007
Note: [a] The date fields of the timestamp structure are ignored.		

For information about what values are valid in an SQL_C_TIMESTAMP structure, see “C data types” on page 210.

When timestamp C data is converted to character SQL data, the resulting character data is in the "yyyy-mm-dd hh:mm:ss [.f. ..]" format.

The driver ignores the length or indicator value when converting data from the timestamp C data types and assumes that the size of the data buffer is the size of the timestamp C data type. The length or indicator value is passed in the StrLen_or_Ind argument in SQLPutData and in the buffer specified with the StrLen_or_IndPtr argument in SQLBindParameter. The data buffer is specified with the DataPtr argument in SQLPutData and the ParameterValuePtr argument in SQLBindParameter.

C to SQL data conversion examples

The following examples illustrate how the driver converts C data to SQL data.

Table 182. C Data to SQL Data

C Data Type	C Data Value	SQL Data Type	Column Length	SQL Data Value	SQLSTATE
SQL_C_CHAR	abcdef\0 [a]	SQL_CHAR	6	abcdef	N/A
SQL_C_CHAR	abcdef\0 [a]	SQL_CHAR	5	abcde	22001
SQL_C_CHAR	1234.56\0 [a]	SQL_DECIMAL	8 [b]	1234.56	N/A
SQL_C_CHAR	1234.56\0 [a]	SQL_DECIMAL	7 [b]	1234.5	22001
SQL_C_CHAR	1234.56\0 [a]	SQL_DECIMAL	4	----	22003
SQL_C_FLOAT	1234.56	SQL_FLOAT	not applicable	1234.56	N/A
SQL_C_FLOAT	1234.56	SQL_INTEGER	not applicable	1234	22001
SQL_C_FLOAT	1234.56	SQL_TINYINT	not applicable	----	22003
SQL_C_TYPE_DATE	1992,12,31 [c]	SQL_CHAR	10	1992-12-31	N/A
SQL_C_TYPE_DATE	1992,12,31 [c]	SQL_CHAR	9	----	22003
SQL_C_TYPE_DATE	1992,12,31 [c]	SQL_TIMESTAMP	not applicable	1992-12-31 00:00:00.0	N/A
SQL_C_TYPE TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	22	1992-12-31 23:45:55.12	N/A
SQL_C_TYPE TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	21	1992-12-31 23:45:55.1	22001
SQL_C_TYPE TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	18	----	22003

Table 182. C Data to SQL Data (continued)

C Data Type	C Data Value	SQL Data Type	Column Length	SQL Data Value	SQLSTATE
Note: [a] "\0" represents a null-termination byte. The null-termination byte is required only if the length of the data is SQL_NTS. [b] In addition to bytes for numbers, one byte is required for a sign and another byte is required for the decimal point. [c] The numbers in this list are the numbers stored in the fields of the SQL_DATE_STRUCT structure. [d] The numbers in this list are the numbers stored in the fields of the SQL_TIMESTAMP_STRUCT structure.					

Appendix F. Scalar functions

This section provides additional information about ODBC scalar functions.

ODBC specifies five types of scalar functions:
- String functions
- Numeric functions
- Time and date functions
- System functions
- Data type conversion functions

A scalar function is a function that returns one value for each row in the query. Functions like SQRT() and ABS() are scalar functions. Functions like SUM() and AVG() are not scalar functions because they return a single value even if they process more than one row.

This section includes tables for each scalar function category. Within each table, functions have been added in ODBC 3.0 to align with SQL-92. Each table also provides the version number when the function was introduced.

ODBC and SQL-92 scalar functions

This topic provides information about ODBC and SQL-92 scalar functions.

Because functions are often data-source-specific, ODBC does not require a data type for return values from scalar functions. To force data type conversion, applications should use the CONVERT scalar function.

Note:

ODBC and SQL-92 classify functions in different ways. ODBC classifies scalar functions by argument type, whereas SQL-92 classifies them by return value. For example, in ODBC, the EXTRACT function is classified as a timedate function because the extract-field argument is a timedate keyword and the extract_source argument is a timedate or interval expression. In SQL-92, however, the EXTRACT function is classified as a numeric scalar function because the return value is numeric.

Applications need to call SQLGetInfo to determine which scalar functions a driver supports. ODBC and SQL-92 information types are available for scalar function classifications. Because ODBC and SQL-92 use different classifications, the information types for the same function may differ between ODBC and SQL-92. For example, to determine support for the EXTRACT function requires SQL_TIMEDATE_FUNCTIONS information type in ODBC and SQL_SQL92_NUMERIC_VALUE_FUNCTIONS information type in SQL-92.

String functions

This topic lists string manipulation functions.

Applications can call SQLGetInfo with the SQL_STRING_FUNCTIONS information type to determine which string functions are supported by a driver.

String Function Arguments

Table 183. String Function Arguments

Arguments denoted as...	Definition
string_exp	These arguments can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, or SQL_LONGVARCHAR.
start, *length* or *count*	These arguments can be a numeric literal or the result of another scalar function, where the underlying data type can be represented as SQL_TINYINT, SQL_SMALLINT, or SQL_INTEGER
character_exp	These arguments are a variable-length character string

The following string functions are 1-based, that is, the first character in the string is character 1, not character 0.

Note: BIT_LENGTH, CHAR_LENGTH, CHARACTER_LENGTH, OCTET_LENGTH, and POSITION string scalar functions were added in ODBC 3.0 to align with SQL-92.

List of String Functions

Table 184. List of String Functions

Function	Description
ASCII(string_exp) (ODBC 1.0)	Returns the ASCII code value of the leftmost character of *string_exp* as an integer.
BIT_LENGTH(string_exp) (ODBC 3.0)	Returns the length in bits of string expression.
CHAR(code) (ODBC 1.0)	Returns the character that has the ASCII code value specified by code. The value of *code* should be between 0 and 255; otherwise, the return value is data source-dependent.
CHAR_LENGTH(string_exp) (ODBC 3.0)	Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as CHARACTER_LENGTH function.)

Table 184. List of String Functions (continued)

Function	Description
CHARACTER_LENGTH(string_exp) (ODBC 3.0)	Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHAR_LENGTH function.)
CONCAT(string_exp1, string_exp2) (ODBC 1.0)	Returns a character string that is the result of concatenating *string_exp2* to *string_exp1*. The resulting string is DBMS-dependent.
DIFFERENCE(string_exp1, string_exp2) (ODBC 2.0)	The function returns the difference between the soundex (see soundex function below) values of two character expressions, as an integer. The integer returned is the number of characters in the soundex values that are the same. The return value ranges from 0 through 4: 0 indicates little or no similarity, and 4 indicates strong similarity or identical values.
INSERT(string_exp1, start, length, string_exp2) (ODBC 1.0)	Returns a character string where *length* characters have been deleted from *string_exp1* beginning at *start* and where *string_exp2* has been inserted into *string_exp*, beginning at *start*.
LCASE(string_exp) (ODBC 1.0)	Returns a string equal to that *string_exp*, with all uppercase characters converted to lowercase.
LEFT(string_exp, count) (ODBC 1.0)	Returns the leftmost *count* of characters of *string_exp*.
LENGTH(string_exp) (ODBC 1.0)	Returns the number of characters in *string_exp*, excluding trailing blanks.

Table 184. List of String Functions (continued)

Function	Description
LOCATE(string_exp1, string_exp2[, start])	Returns the starting position of the first occurrence of *string_exp1* within *string_exp2*. The search for the first occurrence of *string_exp1* begins with the first character position in *string_exp2* unless the optional argument, *start*, is specified. If *start* is specified, the search begins with the character position indicated by the value of *start*. The first character position in *string_exp2* is indicated by the value 1. If *string_exp1* is not found within *string_exp2*, the value 0 is returned. If an application can call the LOCATE scalar function with the *string_exp1*, *string_exp2*, and *start* arguments, the driver returns SQL_FN_STR_LOCATE when SQLGetInfo is called with an option of SQL_STRING_FUNCTIONS. If the application can call the LOCATE scalar function with only the *string_exp1* and *string_exp2* arguments, the driver returns SQL_FN_STR_LOCATE_2 when SQLGetInfo is called with an option of SQL_STRING_FUNCTIONS. Drivers that support calling the LOCATE function with either two or three arguments return both SQL_FN_STR_LOCATE and SQL_FN_STR_LOCATE_2.
LTRIM(string_exp) (ODBC 1.0)	Returns the characters of *string_exp*, with leading blanks removed.
OCTET_LENGTH(string_exp) (ODBC 3.0)	Returns the length in bytes of the string expression. The result is the smallest integer not less than the number of bits divided by 8.
POSITION(character_exp IN character_exp) (ODBC 3.0)	Returns the position of the first character expression in the second character expression. The result is an exact numeric with an implementation-defined precision and a scale of 0.
REPEAT(string_exp, count) (ODBC 1.0)	Returns a character string composed of *string_exp* repeated *count* times.
REPLACE(string_exp1, string_exp2, string_exp3) (ODBC 1.0)	Search *string_exp1* for occurrences of *string_exp2*, and replace with *string_exp3*.
RIGHT(string_exp, count) (ODBC 1.0)	Returns the rightmost *count* of characters of *string_exp*.

Table 184. List of String Functions (continued)

Function	Description
RTRIM(string_exp) (ODBC 1.0)	Returns the characters of *string_exp* with trailing blanks removed.
SOUNDEX(string_exp1) (ODBC 2.0)	Returns a character string containing the phonetic representation of the argument. This function lets you compare words that are spelled differently, but sound alike in English. If you supply a word to Soundex, it returns a 4-character phonetic code used by the U.S.Census Bureau since 1930s.
SPACE(count) (ODBC 2.0)	Returns a character string consisting of *count* spaces.
SUBSTRING(string_exp, start, length) (ODBC 1.0)	Returns a character string that is derived from *string_exp*, beginning at the character position specified by *start* for *length* characters.
TRIM(string_exp)	Returns the characters of *string_exp* with leading blanks and trailing blanks removed.
UCASE(string_exp) (ODBC 1.0)	Returns a string equal to that in *string_exp*, with all lowercase characters converted to uppercase.

Numeric functions

This topic describes numeric functions that are included in the ODBC scalar function set.

Applications can call SQLGetInfo with the SQL_NUMERIC_FUNCTIONS information type to determine which numberic functions are supported by a driver.

Except for ABS, ROUND, TRUNCATE, SIGN, FLOOR, and CEILING (which return values of the same data type as the input parameters), all numeric functions return values of data type SQL_FLOAT.

Numeric Function Arguments

Table 185. Numeric Function Arguments

Arguments denoted as...	Definition
numeric_exp	These arguments can be the name of a column, the result of another scalar function, or a numeric literal, where the underlying data type could be represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, or SQL_DOUBLE

Table 185. Numeric Function Arguments (continued)

Arguments denoted as...	Definition
float_exp	These arguments can be the name of a column, the result of another scalar function, or a numeric literal, where the underlying data type can be represented as SQL_FLOAT.
integer_exp	These arguments can be the name of a column, the result of another scalar function, or a numeric literal, where the underlying data type can be represented as SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, or SQL_BIGINT

List of Numeric Functions

Table 186. List of Numeric Functions

Function	Description
ABS(numeric_exp) (ODBC 1.0)	Returns the absolute value of *numeric_exp*.
ACOS(float_exp) (ODBC 1.0)	Returns the arccosine of *float_exp* as an angle, expressed in radians.
ASIN(float_exp) (ODBC 1.0)	Returns the arcsine of *float_exp* as an angle, expressed in radians.
ATAN(float_exp) (ODBC 1.0)	Returns the arctangent of *float_exp* as an angle, expressed in radians.
ATAN2(float_exp1, float_exp2) (ODBC 2.0)	Returns the arctangent of the x and y coordinates, specified by *float_exp1* and *float_exp2*, respectively, as an angle, expressed in radians.
CEILING(numeric_exp) (ODBC 1.0)	Returns the smallest integer greater than or equal to *numeric_exp*. The return value is of the same data type as the input parameter.
COS(float_exp) (ODBC 1.0)	Returns the cosine of *float_exp*, where *float_exp* is an angle expressed in radians.
COT(float_exp) (ODBC 1.0)	Returns the cotangent of *float_exp*, where *float_exp* is an angle expressed in radians.
DEGREES(numeric_exp) (ODBC 2.0)	Returns the number of degrees converted from *numeric_exp* radians.

Table 186. List of Numeric Functions (continued)

Function	Description
EXP(float_exp) (ODBC 1.0)	Returns the exponential value of *float_exp*.
FLOOR(numeric_exp) (ODBC 1.0)	Returns largest integer less than or equal to *numeric_exp*. The return value is of the same data type as the input parameter.
LOG(float_exp) (ODBC 1.0)	Returns the natural logarithm of *float_exp*.
LOG10(float_exp) (ODBC 2.0)	Returns the base 10 logarithm of *float_exp*.
MOD(integer_exp1, integer_exp2) (ODBC 1.0)	Returns the remainder (modulus) of *integer_exp1* divided by *integer_exp2*.
PI( ) (ODBC 1.0)	Returns the constant value of pi as a floating point value.
POWER(numeric_exp, integer_exp)	Returns the value of *numeric_exp* to the power of *integer_exp*.
RADIANS(numeric_exp) (ODBC 2.0)	Returns the number of radians converted from *numeric_exp* degrees.
ROUND(numeric_exp, integer_exp) (ODBC 2.0)	Returns *numeric_exp* rounded to *integer_exp* places right of the decimal point. If *integer_exp* is negative, *numeric_exp* is rounded to |*integer_exp*| places to the left of the decimal point.
SIGN(numeric_exp) (ODBC 1.0)	Returns an indicator or the sign of *numeric_exp*. If *numeric_exp* is less than zero, -1 is returned. If *numeric_exp* equals zero, 0 is returned. If *numeric_exp* is greater than zero, 1 is returned.
SIN(float_exp) (ODBC 1.0)	Returns the sine of *float_exp*, where *float_exp* is an angle expressed in radians.
SQRT(float_exp) (ODBC 1.0)	Returns the square root of *float_exp*.
TAN(float_exp) (ODBC 1.0)	Returns the tangent of *float_exp*, where *float_exp* is an angle expressed in radians.

Table 186. List of Numeric Functions (continued)

Function	Description
TRUNCATE(numeric_exp, integer_exp) (ODBC 2.0)	Returns *numeric_exp* truncated to *integer_exp* places right of the decimal point. If *integer_exp* is negative, *numeric_exp* is truncated to |*integer_exp*| places to the left of the decimal point.

Time and date functions

This section lists time and date functions that are included in the ODBC scalar function set.

Applications can call SQLGetInfo with the SQL_TIMEDATE_FUNCTIONS information type to determine which time and date functions are supported by a driver.

Time and Data Arguments

Table 187. Time and Data Arguments

Arguments denoted as...	Definition
timestamp_exp	These arguments can be the name of a column, the result of another scalar function, or an *ODBC_time_escape*, *ODBC_date_escape*, or *ODBC_timestamp_escape*, where the underlying data type could be represented as SQL_CHAR, SQL_VARCHAR, SQL_TYPE_TIME, SQL_TYPE_DATE, or SQL_TYPE_TIMESTAMP.
date_exp	These arguments can be the name of a column, the result of another scalar function, or an *ODBC_date_escape* or *ODBC_timestamp_escape*, where the underlying data type could be represented as SQL_CHAR, SQL_VARCHAR, SQL_TYPE_DATE, or SQL_TYPE_TIMESTAMP.
time_exp	These arguments can be the name of a column, the result of another scalar function, or an *ODBC_time_escape* or *ODBC_timestamp_escape*, where the underlying data type could be represented as SQL_CHAR, SQL_VARCHAR, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP

Note: CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP timedate scalar functions were added in ODBC 3.0 to align with SQL-92.

List of Time and Date Functions

Table 188. List of Time and Date Functions

Function	Description
CURRENTTIME [(time_precision)] (ODBC 3.0)	Returns the current local time as a time value. The time_precision argument determines the seconds precision of the returned value.

Table 188. List of Time and Date Functions (continued)

Function	Description
CURRENT_TIMESTAMP [(timestamp_precision)] (ODBC 3.0)	Returns the current local data and local time as a timestamp value. The timestamp_precision argument determines the seconds precision of the returned timestamp.
CURDATE( ) (ODBC 1.0)	Returns the current date.
CURTIME( ) (ODBC 1.0)	Returns the current local time.
DAYNAME(date_exp) (ODBC 2.0)	Returns a character string containing the data source-specific name of the day (for example, Sunday, through Saturday or Sun. through Sat. for a data source that uses English, or Sonntag through Samstag for a data source that uses German) for the day portion of *date_exp*.
DAYOFMONTH(date_exp) (ODBC 1.0)	Returns the day of the month in *date_exp* as an integer value in the range of 1-31.
DAYOFWEEK(date_exp) (ODBC 1.0)	Returns the day of the week based on the week field in *date_exp* as an integer value in the range of 1-7, where 1 represents Sunday.
DAYOFYEAR(date_exp) (ODBC 1.0)	Returns the day of the year based on the year field in *date_exp* as an integer value in the range of 1-366.

EXTRACT(extract_field FROM extract_source)
 	
(ODBC 3.0)
	Returns the *extract_field* portion of the *extract_source*. The extract_source argument is a datetime or interval expression. The *extract_field* argument can be one of the following keywords"
	`YEAR`
	`MONTH`
	`DAY`
	`HOUR`
	`MINUTE`
	`SECOND`
	
	The precision of the returned value is implementation-defined. The scale is 0 unless SECOND is specified, in which case the scale is not less than the fractional seconds precision of the extract_source field.

HOUR(time_exp) (ODBC 1.0)	Returns the hour based on the hour field in *time_exp* as an integer value in the range of 0-23.
MINUTE(time_exp) (ODBC 1.0)	Returns the minute based on the minute field in *time_exp* as an integer value in the range of 0-59.

Table 188. List of Time and Date Functions (continued)

Function	Description
MONTH(date_exp) (ODBC 1.0)	Returns the month based on the month field in *date_exp* as an integer value in the range of 1-12.
MONTHNAME(date_exp) (ODBC 2.0)	Returns a character string containing the data source-specific name of the month (for example, January through December or Jan. through Dec. for a data source that uses English, or Januar through Dezember for a data source that uses German) for the month portion of *date_exp*.
NOW( ) (ODBC 1.0)	Returns current date and time as a timestamp value.
QUARTER(date_exp) (ODBC 1.0)	Returns the quarter in *date_exp* as an integer value in the range of 1- 4, where 1 represents January 1 through March 31.
SECOND(time_exp) (ODBC 1.0)	Returns the second in *time_exp* as an integer value in the range of 0-59.

Table 188. List of Time and Date Functions (continued)

Function	Description

TIMESTAMPADD(interval, integer_exp, timestamp_exp)
 	
(ODBC 2.0)
	Returns the timestamp calculated by adding *integer_exp* intervals of type interval to *timestamp_exp*. Valid values of interval are the following keywords:
	`SQL_TSI_FRAC_SECOND`
	`SQL_TSI_SECOND`
	`SQL_TSI_MINUTE`
	`SQL_TSI_HOUR`
	`SQL_TSI_DAY`
	`SQL_TSI_WEEK`
	`SQL_TSI_MONTH`
	`SQL_TSI_QUARTER`
	`SQL_TSI_YEAR`
	
	where fractional seconds are expressed in billionths of a second (nanoseconds). For example, the following SQL statement returns the name of each employee and his or her one-year anniversary date:
	`SELECT NAME, {fn`
	`TIMESTAMPADD(SQL_TSI_YEAR, 1,`
	`HIRE_DATE)} FROM`
	`EMPLOYEES`
	
	If *timestamp_exp* is a time value and interval specifies day, weeks, months, quarters, or years, the date portion of *timestamp_exp* is set to the current date before calculating the resulting timestamp.
	
	If *timestamp_exp* is a date value and interval specifies fractional seconds, seconds, minutes, or hours, the time portion of *timestamp_exp* is set to 0 before calculating the resulting timestamp.
	
	An application determines which intervals a data source supports by calling SQLGetInfo with the SQL_TIMEDATE_ADD_INTERVALS option.

Table 188. List of Time and Date Functions (continued)

Function	Description

TIMESTAMPDIFF(interval, timestamp_exp1, timestamp_exp2)
 	
(ODBC 2.0)
	Returns the number of unit intervals (as integers) of type *interval* between *timestamp_exp1* and *timestamp_exp2*.
	
	If an application relies on the old TIMESTAMPDIFF semantics, the old behavior can be emulated by the following configuration setting in the SQL section of the `solid.ini` file.
	
	`[SQL]`
	`EmulateOldTIMESTAMPDIFF=YES`
	
	Note that the old semantics returns the integer number of intervals of type interval by which *timestamp_exp2* is greater than *timestamp_exp1*.
	
	Valid values of interval are the following keywords:
	
	`SQL_TSI_FRAC_SECOND`
	`SQL_TSI_SECOND`
	`SQL_TSI_MINUTE`
	`SQL_TSI_HOUR`
	`SQL_TSI_DAY`
	`SQL_TSI_WEEK`
	`SQL_TSI_MONTH`
	`SQL_TSI_QUARTER`
	`SQL_TSI_YEAR`
	
	where fractional seconds are expressed in billionths of a second (nanoseconds). For example, the following SQL statement returns the name of each employee and the number of years they have been employed:
	
	`SELECT NAME, {fn`
	`TIMESTAMPDIFF(SQL_TSI_YEAR,`
	`{fn CURDATE()}, HIRE_DATE)}`
	`FROM EMPLOYEES`
	
	If either timestamp expression is a time value and interval specifies days, weeks, months, quarters, or years, the date portion of that timestamp is set to the current date before calculating the difference between the timestamps.
	
	If either timestamp expression is a date value and interval specifies fractional seconds, seconds, minutes, or hours, the time portion of that timestamp is set to 0 before calculating the difference between the timestamps.
	
	An application determines which intervals a data source supports by calling SQLGetInfo with the SQL_TIMEDATE_DIFF_INTERVALS option.

WEEK(date_exp) (ODBC 1.0)	Returns the week of the year based on the week field in *date_exp* as an integer value in the range of 1-53.
YEAR(date_exp) (ODBC 1.0)	Returns the year based on the year field in *date_exp* as an integer value. The range is data source-dependent.

System functions

This section lists system functions that are included in the ODBC scalar function set.

Applications can call SQLGetInfo with the SQL_SYSTEM_FUNCTIONS information type to determine which string functions are supported by a driver.

System Functions Arguments

Table 189. System Function Arguments

Arguments denoted as...	Definition
exp	These arguments can be the name of a column, the result of another scalar function, or a literal, where the underlying data type could be represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, SQL_DOUBLE, SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP.
value	These arguments can be a literal constant, where the underlying data type can be represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, SQL_DOUBLE, SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP.
integer_exp	These arguments can be the name of a column, the result of another scalar function, or a numeric literal, where the underlying data type can be represented as SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, or SQL_BIGINT

Values returned are represented as ODBC data types

List of System Functions

Table 190. List of System Functions

Function	Description
DATABASE( ) (ODBC 1.0)	Returns the name of the database corresponding to the connection handle. (The name of the database is also available by calling SQLGetConnectOption with the SQL_CURRENT_QUALIFIER connection option.)
IFNULL(exp, value) (ODBC 1.0)	If exp is null, value is returned. If exp is not null, exp is returned. The possible data type(s) of value must be compatible with the data type of exp
USER( ) (ODBC 1.0)	Returns the user's name in the DBMS. (The user's authorization name is also available via SQLGetInfo by specifying the information type: SQL_USER_NAME.) This can be different from the login time.

Explicit data type conversion

Explicit data type conversion is specified in terms of SQL data type definitions.

The ODBC syntax for the explicit data type conversion function does not restrict conversions. The validity of specific conversions of one data type to another data type is dependent on each driver-specific implementation. The driver, as it translates the ODBC syntax into the native syntax, reject those conversions that, although legal in the ODBC syntax, are not supported by the data source. Applications can call the ODBC function SQLGetInfo to inquire about conversions supported by the data source.

The format of the CONVERT function is:

CONVERT(*value_exp*, *data_type*)

The function returns the value specified by value_exp converted to the specified data_type, where data_type is one of the following keywords:

- SQL_BIGINT
- SQL_SMALLINT
- SQL_BINARY
- SQL_DATE
- SQL_CHAR
- SQL_TIME
- SQL_DECIMAL
- SQL_TIMESTAMP
- SQL_DOUBLE
- SQL_TINYINT
- SQL_FLOAT
- SQL_VARBINARY
- SQL_INTEGER
- SQL_VARCHAR
- SQL_LONGVARBINARY
- SQL_WCHAR
- SQL_LONGVARCHAR
- SQL_WLONGVARCHAR
- SQL_NUMERIC
- SQL_WVARCHAR
- SQL_REAL

The ODBC syntax for the explicit data type conversion function does not support specification of conversion format. If specification of explicit formats is supported by the underlying data source, a driver must specify a default value or implement format specification.

The argument value_exp can be a column name, the result of another scalar function, or a numeric or string literal. The following example converts the output of the CURDATE scalar function to a character string:

```
{ fn CONVERT( { fn CURDATE() }, SQL_CHAR) }
```

ODBC does not require a data type for return values from scalar functions (because the functions are often data source-specific); applications should use the CONVERT scalar function whenever possible to force data type conversion.

The following two examples illustrate the use of the CONVERT function. These examples assume the existence of a table called EMPLOYEES, with an EMPNO column of type SQL_SMALLINT and an EMPNAME column of type SQL_CHAR.

If an application specifies the following:

```
SELECT EMPNO FROM EMPLOYEES WHERE {fn CONVERT(EMPNO,SQL_CHAR)}LIKE '1%'
```

solidDB ODBC driver translates the request to:

```
SELECT EMPNO FROM EMPLOYEES WHERE CONVERT_CHAR(EMPNO) LIKE '1%'
```

SQL-92 CAST function

The ODBC CONVERT function has an equivalent function in SQL-92: the CAST function.

The syntax for these equivalent functions is as follows::

```
{ fn CONVERT (value_exp, data_type)} /* ODBC */
CAST (value_exp AS data_type) /* SQL 92 */
```

Support for the CAST function is at the FIPS Transitional level. For details on data type conversion in the CAST function, see the SQL-92 specification.

To determine application support for the CAST function, call SQLGetInfo with the SQL_SQL_CONFORMANCE information type. The CAST function is supported if the return value for the information type is:

- SQL_SC_FIPS127_2_TRANSITIONAL
- SQL_SC_SQL92_INTERMEDIATE
- SQL_SC_SQL92_FULL

If the return value is SQL_SC_ENTRY or 0, call SQLGetInfo with the SQL_SQL92_VALUE_EXPRESSIONS information type. If the SQL_SVE_CAST bit is set, the CAST function is supported.

Appendix G. Timeout controls

In solidDB, some actions can get timed out. A timeout can be activated by the main server, the client drivers, the Primary or Secondary server, or the Master or Replica server.

Timeouts have factory default values and they can usually be set with different .ini parameters. Some startup defaults can be dynamically changed with different controls, by using SQL, or by using the driver interfaces and connection string parameters.

Client timeouts

Timeouts related to the database client are introduced in this topic.

Login timeout

This timeout refers to the number of seconds the driver waits for the login (SQLConnect) to succeed. The default value is driver-dependent. If the value (or ValuePtr in ODBC) is 0, the timeout is disabled and a connection attempt will wait indefinitely. If the specified timeout exceeds the maximum login timeout in the data source, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed).

This timeout applies for the TCP protocol only.

Table 191. Login timeouts

INI parameter	Overridden with SQL	Driver	Connection string
		ODBC: SQL_ATTR_LOGIN_TIMEOUT (in seconds) SQL_ATTR_LOGIN_TIMEOUT_MS (in milliseconds, non-standard) JDBC: Method (JDBC 2.0) DriverManger.setLoginTimeout(seconds); Connection property (non-standard) "solid_login_timeout_ms" (milliseconds)	-c milliseconds

Table 191. Login timeouts (continued)

INI parameter	Overridden with SQL	Driver	Connection string
(client-side) [Com] ConnectTimeout (in milliseconds) or: Connect -c option (in milliseconds)			

Timeout error code and message:

ODBC:

HYT00, Timeout expired

Connection Timeout

This timeout refers to the number of seconds (or milliseconds) the driver waits for any request on the connection to complete. This timeout is not associated with the query execution or login. Upon timeout, the driver disconnects from the solidDB server.

The driver returns SQLSTATE HYT00 (Timeout expired) if it is possible to time out in a situation not associated with query execution or login. If the value (or ValuePtr in ODBC) is 0 (the default value), there is no timeout.

This timeout applies to all ODBC functions (ODBC 3.5 specifications) except:

```
SQLDrivers
SQLDataSources
SQLGetEnvAttr
SQLSetEnvAttr
```

Table 192. Connection Timeout

INI parameter	Overridden with SQL	Driver	Connection string
(server-side) [Com] Listen -r option (in milliseconds) (client-side) [Com] ClientReadTimeout (in milliseconds) or: Connect (-r option) (in milliseconds)		ODBC: SQL_ATTR_CONNECTION_TIMEOUT (in seconds) SQL_ATTR_CONNECTION_TIMEOUT_MS (in milliseconds, non-standard) JDBC: Non-standard: Connection property "solid_connection_timeout_ms" (milliseconds) or method: SolidConnection.setConnectionTimeout() (milliseconds)	-r milliseconds

Note: This timeout has also been implemented on the server, which means that the server will cancel the outstanding request and disconnect the client.

Timeout error code and message:

ODBC:

HYT01, Connection timeout expired

See also:

```
SOLID Server Error 14518:
Connection to the server is broken, connection lost.

SOLID Communication Error 21328 and SOLID Session Error 20024:
Timeout while resolving host name.

SOLID Communication Error 21329 and SOLID Session Error 20025:
Timeout while connecting to a remote host.
```

Query timeout

This timeout refers to the number of seconds the driver waits for an SQL statement to execute. If the value (or ValuePtr in ODBC) is 0 (the default value), there is no timeout.

If the specified timeout exceeds the maximum timeout in the data source, or if the specified timeout is smaller than the minimum timeout, SQLSetStmtAttr substitutes that value and returns SQLSTATE 01S02 (Option value changed).

This timeout applies to the ODBC functions (ODBC 3.5 specifications) as follows:

SQLBrowseConnect

SQLBulkOperations

SQLColumnPrivileges

SQLColumns

SQLConnect

SQLDriverConnect

SQLExecDirect

SQLExecute

SQLExtendedFetch

SQLForeignKeys

SQLGetTypeInfo

SQLParamData

SQLPrepare

SQLPrimaryKeys

SQLProcedureColumns

SQLProcedures

SQLSetPos

SQLSpecialColumns

SQLStatistics

SQLTablePrivileges

SQLTables

Note: The application need not call SQLCloseCursor to reuse the statement if a SELECT statement timed out. The query timeout set in this statement attribute is valid in both synchronous and asynchronous modes.

Table 193. Query Timeout

INI parameter	Overridden with SQL	Driver	Connection string
		ODBC: SQL_ATTR_QUERY_TIMEOUT (in seconds) SQL_ATTR_QUERY_TIMEOUT_MS (in milliseconds, non-standard)	

Timeout error code and message:

ODBC:

HYT00, Timeout expired

Server timeouts

Timeouts related to the database server are introduced in this topic.

SQL Statement Execution Timeout

The server can control the amount of time spent on the execution of one SQL statement. When the time expires, the server terminates the statements and returns a corresponding error code. This timeout applies to the following calls (ODBC 3.5 specifications):

- SQLExecute()
- SQLExecDirect()
- SQLPrepare()
- SQLForeignKeys()
- SQLColumns()
- SQLProcedureColumns()
- SQLSpecialColumns()
- SQLStatistics()
- SQLPrimaryKeys()
- SQLProcedures()
- SQLTables()
- SQLTablePrivileges()
- SQLColumnPrivileges()
- SQLGetTypeInfo()

The timeout also applies to the corresponding JDBC calls.

Table 194. SQL statement execution timeouts

INI parameter	Overridden with SQL	Driver	Connection string
	SET STATEMENT MAXTIME minutes	ODBC: SQL_ATTR_QUERY_TIMEOUT (in seconds) SQL_ATTR_QUERY_TIMEOUT_MS (in milliseconds, non-standard) JDBC: statement.setQueryTimeout()	

Timeout error code and message:

HYT00, Timeout expired

See also:

```
SOLID Server Error 14518:
Connection to the server is broken, connection lost.

SOLID Server Error 14529:
The operation timed out.
```

Lock wait timeout

This timeout specifies the time in seconds (or milliseconds) that the engine waits for a lock to be released. When the timeout interval is reached, solidDB terminates the timed-out transaction. The default value is 30 seconds, and the parameter access mode is read/write.

Lock wait timeout is used in deadlock resolution. In that case, the oldest transaction participating in a deadlock is aborted.

Table 195. Lock wait timeout

INI parameter	Overridden with SQL	Driver	Connection string
[General] LockWaitTimeOut seconds	SET LOCK TIMEOUT {seconds | milliseconds MS}		

Timeout error code and message:

```
SOLID Database Error 10006:
Concurrency conflict, two transactions updated or deleted the same row.
```

Optimistic lock timeout

This timeout specifies the optimistic lock timeout. Optimistic lock is an additional lock that can be enacted in order to ensure that SELECT FOR UPDATE will always lead to successful updates, in the optimistic concurrency mode. The default is zero whereby no optimistic lock is used, and a transaction may be aborted after each statement, as a result of early transaction validation. When the timeout is set to a non-zero value, SELECT FOR UPDATE will wait until the lock is obtained, or it is timed-out and aborted. When set, the timeout affects also all DELETE and UPDATE statements.

Table 196. Optimistic lock wait timeout

INI parameter	Overridden with SQL	Driver	Connection string
	SET OPTIMISTIC LOCK TIMEOUT {seconds | milliseconds MS}		

Timeout error code and message:

```
SOLID Database Error 10006:
Concurrency conflict, two transactions updated or deleted the same row.
```

Table lock wait timeout

Occasionally, the transaction will acquire an exclusive lock to a table. This may be result of a lock escalation, an attempt to execute the ALTER TABLE statement, or as a side effect of some advanced replication commands. If there is a table-level conflict, this setting provides the transaction's wait period until the exclusive or shared lock is released. The unit is seconds, the default value is 30 seconds, and the parameter access mode is read/write.

To be more specific, table level locks are used when the PESSIMISTIC keyword is explicitly provided in the following commands:

```
IMPORT SUBSCRIPTIONMESSAGE message_name EXECUTE
(only with NO EXECUTE option)
MESSAGE message_name FORWARD
MESSAGE message_name GET REPLY
DROP SUBSCRIPTION.
```

Table 197. Table Lock Wait Timeout

INI parameter	Overridden with SQL	Driver	Connection string
[General] TableLockWaitTimeout seconds			

Timeout error code and message:

```
SOLID Database Error 10006:
Concurrency conflict, two transactions updated or deleted the same row.
```

Transaction idle timeout

This timeout specifies the time in minutes after an idle transaction is aborted; a negative or zero value means infinite. The unit is minutes, the default value is 120 minutes, and the access mode is read/write.

Table 198. Transaction Idle Timeout

INI parameter	Overridden with SQL	Driver	Connection string
[Srv] AbortTimeOut			

Timeout error code and message:

```
SOLID Database Error 10026:
 Transaction is timed out.
```

Connection idle timeout

This timeout specifies the continuous idle time in minutes (or seconds/milliseconds in a statement) after which a connection is dropped (by the server); negative or zero value indicates an infinite value. The parameter unit is minutes, the default value is 480 minutes, and the access mode is read/write.

Table 199. Connection Idle Timeout

INI parameter	Overridden with SQL	Driver	Connection string
[Srv] ConnectTimeOut (minutes)	SET IDLE TIMEOUT {seconds | milliseconds MS}	JDBC: Connection property (non-standard): "solid_idle_timeout_min"	

Timeout error code and message:

```
SOLID Communication Error 21308:
Connection is broken (protocol read/write
operation failed with code internal code).
```

See also the `solmsg.out` file.

If the SET IDLE TIMEOUT has been set and the transaction is idle for the given period, the error below is given:

```
SOLID Database Error 10026:
Transaction is timed out
```

HotStandby timeouts

Timeouts related to the HotStandby server are introduced in this topic.

Connect Timeout

By specifying a connect timeout value, you can set the maximum time in milliseconds that a HotStandby connect operation waits for a connection to a remote machine. The ConnectTimeout parameter is only used with the following subset of administration commands:

```
hotstandby connect
hotstandby switch primary
hotstandby switch secondary
```

The unit is milliseconds, the default value is 3000, and the access mode is read/write.

Table 200. Connect timeout

INI parameter	Overridden with SQL	Driver	Connection string
[HotStandby] ConnectTimeout milliseconds			

Ping timeout

This parameter specifies how long a server waits before concluding that the other server is down or inaccessible. The unit is milliseconds, the default value is 4000, and the access mode is read/write.

Table 201. Ping timeout

INI parameter	Overridden with SQL	Driver	Connection string
[HotStandby] PingTimeout milliseconds			

Appendix H. Client-side configuration parameters

The client-side configuration parameters are stored in the `solid.ini` configuration file and are read when the client starts.

Generally, the factory value settings offer the best performance and operability, but in some special cases modifying a parameter will improve performance. You can change the parameters by editing the configuration file `solid.ini`.

The parameter values set in the client side configuration file come to effect each time an application issues a call to the SqlConnect ODBC function. If the values are changed in the file during the program's run time, they affect the connections established thereafter.

Setting client-side parameters through the solid.ini configuration file

This topic provides details about the solid.ini configuration file.

When the solidDB is started, it attempts to open the configuration file `solid.ini`. If the file does not exist, solidDB will use the factory values for the parameters. If the file exists, but a value for a particular parameter is not set in the `solid.ini` file, solidDB will use a factory value for that parameter. The factory values may depend on the operating system you are using.

By default, the client looks for the `solid.ini` file in the current working directory, which is normally the directory from which you started the client. When searching for the file, the solidDB uses the following precedence (from high to low):

- location specified by the SOLIDDIR environment variable (if this environment variable is set)
- current working directory

Rules for formatting the client-side solid.ini file

When you format the client-side `solid.ini` file, the same rules apply as for the server-side `solid.ini` file. For more information, refer to section "Rules for Formatting the solid.ini File" in *solidDB Administration Guide*.

Client-side solid.ini file

```
[Com]
;use this connect string of no data source given
Listen = tcp host1.acme.com 1315

[Client]
;at SQLConnect, timeout after this time (ms)
ConnectTimeout = 5000

;at any ODBC network request, timeout after this time (ms)
ClientReadTimeout = 10000

[DataSources]
Primary_Server = tcp irix1 1315, The Primary Server
Secondary_Server = tcp irix2 1315, The Secondary Server
```

Descriptions of client-side configuration parameters

This topic describes the three section of the`solid.ini` file. The sections are the communication, data sources, and client.

Communication section

Table 202. Communication parameters

[Com]	Description	Factory Value
ClientReadTimeout	This parameter defines the connection (or read) timeout in milliseconds. A network request fails if no response is received during the time specified. The value 0 sets the timeout to infinite. This value can be overridden with the connect string option -r and, further on, with the ODBC attribute SQL_ATTR_CONNECTION_TIMEOUT. Note: applies for the TCP protocol only.	60 000
Connect	The Connect parameter defines the default network name (connect string) for a client to connect to when it establishes a connection to a server. This value is used when the SQLConnect() call is issued with an empty data source name.	tcp localhost 1964
ConnectTimeout	The ConnectTimeout parameter defines the login timeout in milliseconds. This value can be overridden with the connect string option -c and, further on, with the ODBC attribute SQL_ATTR_LOGIN_TIMEOUT. Note: applies for the TCP protocol only.	OS-specific
ODBCHandleValidation	The ODBCHandleValidation parameter switches ODBC handle validation on/off. See also in section "ODBC Handle Validation" in *solidDB Programmer Guide* for more information on the `SQL_ATTR_HANDLE_VALIDATION` ODBC attribute.	No
Trace	If this parameter is set to yes, trace information on network messages for the established network connection is written to a file specified with the TraceFile parameter. The factory value for the TraceFile parameter is `soltrace.out`.	no

Table 202. Communication parameters (continued)

[Com]	Description	Factory Value
TraceFile	If the Trace parameter is set to yes, trace information on network messages is written to a file specified with this TraceFile parameter.	`soltrace.out` (written to the current working directory of the server or client depending on which end the tracing is started)

Data sources

Table 203. Data source parameters

[Data Sources]	Description	Factory Value	Access Mode
logical name = network name, Description	These parameters can be used to give a logical name to a solidDB server in a `solid.ini` file of the client application. For details, read section *Logical Data Source Names* in *solidDB Administration Guide*.		N/A

Client

Table 204. Client parameters

[Client]	Description	Factory Value
ExecRowsPerMessage	This parameter specifies how many result rows are sent (pre-fetched) to the client driver in response to the SQLExecute call with a SELECT statement. The result rows are subsequently returned to the application with the first SQLFetch calls issued by the application. The default value of 2 allows for pre-fetching of single-row results. If your SELECT statements usually return larger number of rows, setting this to an appropriate value can improve performance significantly. See also the RowsPerMessage configuration parameter.	decided by the server
NoAssertMessages	This parameter is relevant to the Windows platform only. If set to Yes, the Windows run-time error dialog is not shown.	No
ODBCCharBinding	If set to UTF-8, ODBC-applications are allowed to store and retrieve UNICODE data in UTF-8 encoded format.	Raw

Table 204. Client parameters (continued)

[Client]	Description	Factory Value
RowsPerMessage	Specifies the number of rows returned from the server in one network message when an SQLFetch call is executed (and there are no pre-fetched rows). See also the ExecRowsPerMessage configuration parameter.	decided by the server
StatementCache	Statement cache is an internal memory storing a few previously prepared SQL statements. With this parameter, you can set the number of cached statements per session.	6

Index

T

U

V

W

Y

Notices

This product is protected by U.S. patents 6144941, 7136912, 6970876, 7139775, 6978396, 7266702, 7406489, 7502796, and 7587429.

This product is assigned the U.S. Export Control Classification Number ECCN=5D992b.

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, ibm.com®, Solid, solidDB, InfoSphere™, DB2®, Informix®, and WebSphere are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

Printed in USA

SC23-9825-03