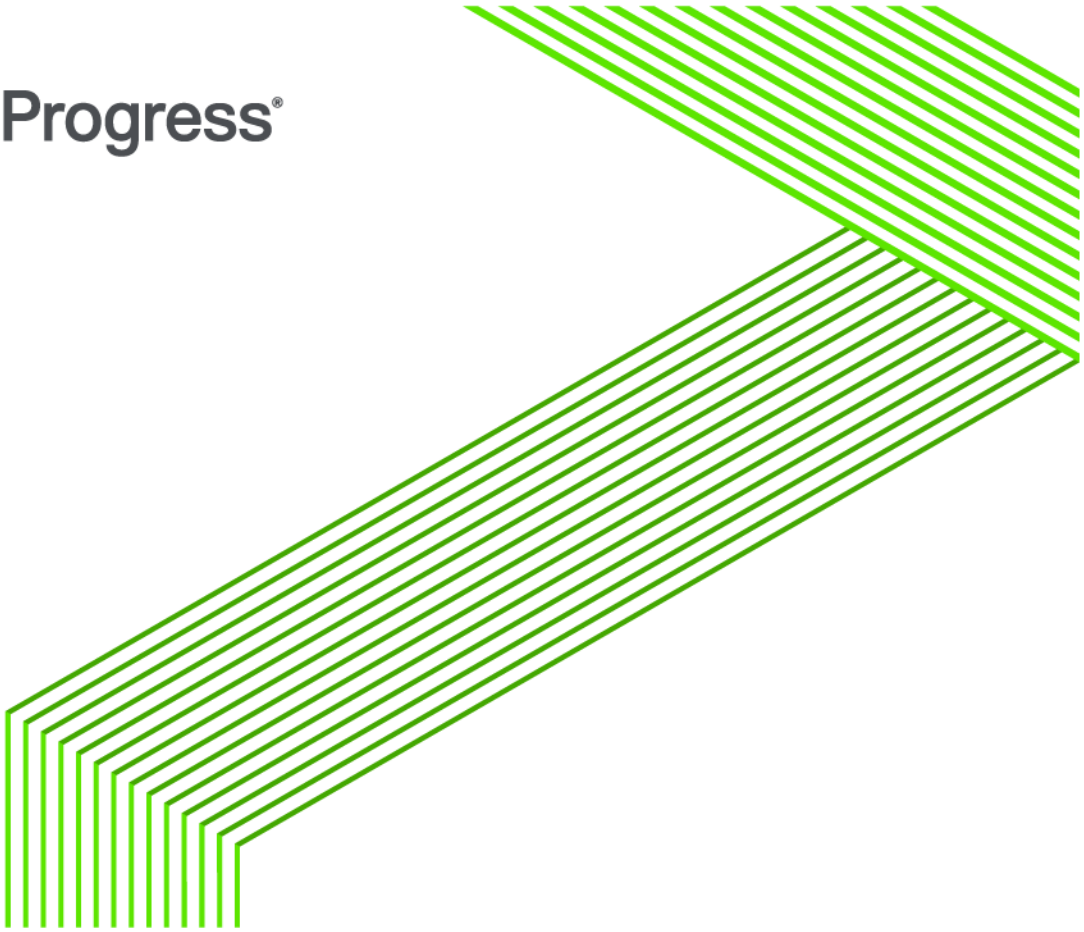


Progress DataDirect for ODBC Drivers

Reference

December 2020

Copyright

Updated: 2021/01/08

Table of Contents

1

Welcome to the Progress DataDirect for ODBC Drivers Reference

This guide provides general information for Progress DataDirect for ODBC drivers, including troubleshooting tips, descriptions of advanced features, and optimizing ODBC applications. The content of this guide applies to all ODBC drivers unless otherwise noted.

The reference acts as a compliment to the driver user's guides, which provide detailed instructions on configuring and using drivers. For complete driver documentation sets, visit the Progress Documentation Hub: https://docs.progress.com/bundle/datadirect-connectors/page/DataDirect-Connectors-by-data-source.html.

Note: This reference refers the reader to Web pages using URLs for more information about specific topics, including Web URLs not maintained by Progress DataDirect. Because it is the nature of Web content to change frequently, Progress DataDirect can guarantee only that the URLs in this reference were correct at the time of publishing.

For details, see the following topics:

What is ODBC?

The Open Database Connectivity (ODBC) interface by Microsoft allows applications to access data in database management systems (DBMS) using SQL as a standard for accessing the data. ODBC permits maximum interoperability, which means a single application can access different DBMS. Application end users can then add ODBC database drivers to link the application to their choice of DBMS.

The ODBC interface defines:

- A library of ODBC function calls of two types:
  - Extended functions that support additional functionality, including scrollable cursors
  - Core functions that are based on the X/Open and SQL Access Group Call Level Interface specification
- SQL syntax based on the X/Open and SQL Access Group SQL CAE specification (1992)
- A standard set of error codes
- A standard way to connect and logon to a DBMS
- A standard representation for data types

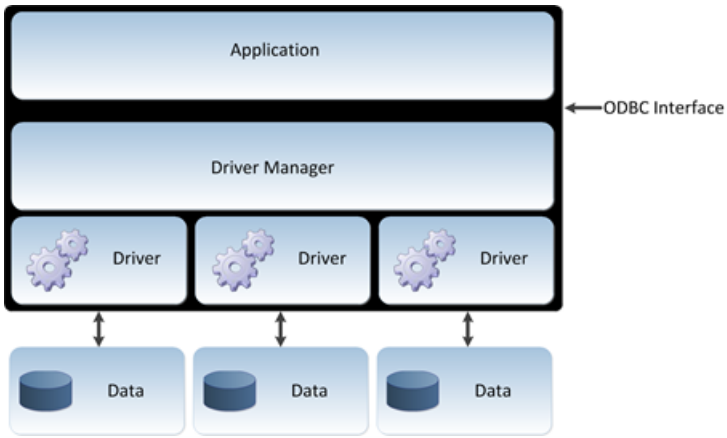
The ODBC solution for accessing data led to ODBC database drivers, which are dynamic-link libraries on Windows and shared objects on UNIX and Linux. These drivers allow an application to gain access to one or more data sources. ODBC provides a standard interface to allow application developers and vendors of database drivers to exchange data between applications and data sources.

How does it work?

The ODBC architecture has four components:

- An application, which processes and calls ODBC functions to submit SQL statements and retrieve results
- A Driver Manager, which loads drivers for the application
- A driver, which processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application
- A data source, which consists of the data to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS

The following figure shows the relationship among the four components:



Why do application developers need ODBC?

Using ODBC, you, as an application developer can develop, compile, and ship an application without targeting a specific DBMS. In this scenario, you do not need to use embedded SQL; therefore, you do not need to recompile the application for each new environment.

2

Troubleshooting

This part guides you through troubleshooting Progress DataDirect for ODBC drivers. It provides you with solutions to common problems and documents error messages that you may receive.

For details, see the following topics:

- Diagnostic tools
- Error messages
- Troubleshooting issues

Diagnostic tools

This chapter discusses the diagnostic tools you use when configuring and troubleshooting your ODBC environment.

ODBC trace

ODBC tracing allows you to trace calls to ODBC drivers and create a log of the traces.

Creating a trace Log

Creating a trace log is particularly useful when you are troubleshooting an issue.

To create a trace log:

1. Enable tracing (see "Enabling tracing" for more information).
2. Start the ODBC application and reproduce the issue.
3. Stop the application and turn off tracing.
4. Open the log file in a text editor and review the output to help you debug the problem.

For a complete explanation of tracing, refer to the following Progress DataDirect Knowledgebase document:

http://knowledgebase.progress.com/articles/Article/3049

See also

Enabling tracing on page 14

Enabling tracing

Progress DataDirect provides a tracing library that is enhanced to operate more efficiently, especially in production environments, where log files can rapidly grow in size. The DataDirect tracing library allows you to control the size and number of log files.

On Windows, you can enable tracing through the Tracing tab of the ODBC Data Source Administrator.

On UNIX and Linux, you can enable tracing by directly modifying the `[ODBC]` section in the system information (`odbc.ini`) file.

On macOS, you can also enable tracing through the Tracing tab of the iODBC Data Source Administrator.

Windows ODBC Administrator

On Windows, open the ODBC Data Source Administrator and select the Tracing tab. To specify the path and name of the trace log file, type the path and name in the Log File Path field or click **Browse** to select a log file. If no location is specified, the trace log resides in the working directory of the application you are using.

Click **Select DLL** in the Custom Trace DLL pane to select the DataDirect enhanced tracing library, *xx*`trc`*yy*`.dll`, where *xx* represents either `iv` (32-bit version) or `dd` (64-bit version), and *yy* represents the driver level number, for example, `ivtrc28.dll`. The library is installed in the `\Windows\System32` directory.

After making changes on the Tracing tab, click **Apply** for them to take effect.

Enable tracing by clicking **Start Tracing Now**. Tracing continues until you disable it by clicking **Stop Tracing Now**. Be sure to turn off tracing when you are finished reproducing the issue because tracing decreases the performance of your ODBC application.

When tracing is enabled, information is written to the following trace log files:

- Trace log file (*`trace_filename`*`.log`) in the specified directory.
- Trace information log file (*`trace_filename`*`INFO.log`). This file is created in the same directory as the trace log file and logs the following SQLGetInfo information:
  - SQL_DBMS_NAME
  - SQL_DBMS_VER
  - SQL_DRIVER_NAME
  - SQL_DRIVER_VER
  - SQL_DEFAULT_TXN_ISOLATION

The DataDirect enhanced tracing library allows you to control the size and number of log files. The file size limit of the log file (in KB) is specified by the Windows Registry key ODBCTraceMaxFileSize. Once the size limit is reached, a new log file is created and logging continues in the new file until it reaches its file size limit, after which another log file is created, and so on.

The maximum number of files that can be created is specified by the Registry key ODBCTraceMaxNumFiles. Once the maximum number of log files is created, tracing reopens the first file in the sequence, deletes the content, and continues logging in that file until the file size limit is reached, after which it repeats the process with the next file in the sequence. Subsequent files are named by appending sequential numbers, starting at 1 and incrementing by 1, to the end of the original file name, for example, `SQL1.LOG`, `SQL2.LOG`, and so on.

The default values of ODBCTraceMaxFileSize and ODBCTraceMaxNumFiles are `102400` KB and `10`, respectively. To change these values, add or modify the keys in the following Windows Registry section:

`[HKEY_CURRENT_USER\SOFTWARE\ODBC\ODBC.INI\ODBC]`

Warning: Do not edit the Registry unless you are an experienced user. Consult your system administrator if you have not edited the Registry before.

Edit each key using your values and close the Registry.

System information (odbc.ini) file

UNIX®

The `[ODBC]` section of the system information file includes several keywords that control tracing:

```
Trace=[0 | 1]
TraceFile=trace_filename
TraceDll=ODBCHOME/lib/xxtrcyy.zz
ODBCTraceMaxFileSize=file_size
ODBCTraceMaxNumFiles=file_number
TraceOptions=0
```

where:

`Trace=[0 | 1]`

Allows you to enable tracing by setting the value of Trace to `1`. Disable tracing by setting the value to `0` (the default). Tracing continues until you disable it. Be sure to turn off tracing when you are finished reproducing the issue because tracing decreases the performance of your ODBC application.

`TraceFile=`*`trace_filename`*

Specifies the path and name of the trace log file. If no path is specified, the trace log resides in the working directory of the application you are using.

`TraceDll=ODBCHOME/lib/`*`xxtrcyy.zz`*

Specifies the library to use for tracing. The driver installation includes a DataDirect enhanced library to perform tracing, *`xx`*`trcyy.zz`, where *`xx`* represents either `iv` (32-bit version) or `dd` (64-bit version), *`yy`* represents the driver level number, and *`zz`* represents either `so` or `sl`. For example, `ivtrc28.so` is the 32-bit version of the library. To use a custom shared library instead, enter the path and name of the library as the value for the TraceDll keyword.

The DataDirect enhanced tracing library allows you to control the size and number of log files with the ODBCTraceMaxFileSize and ODBCTraceMaxNumFiles keywords.

`ODBCTraceMaxFileSize=file_size`

The ODBCTraceMaxFileSize keyword specifies the file size limit (in KB) of the log file. Once this file size limit is reached, a new log file is created and logging continues in the new file until it reaches the file size limit, after which another log file is created, and so on. The default is `102400`.

`ODBCTraceMaxNumFiles=file_number`

The ODBCTraceMaxNumFiles keyword specifies the maximum number of log files that can be created. The default is `10`. Once the maximum number of log files is created, tracing reopens the first file in the sequence, deletes the content, and continues logging in that file until the file size limit is reached, after which it repeats the process with the next file in the sequence. Subsequent files are named by appending sequential numbers, starting at 1 and incrementing by 1, to the end of the original file name, for example, `odbctrace1.out, odbctrace2.out`, and so on.

`TraceOptions=[0 | 1 | 2 | 3]`

The ODBCTraceOptions keyword specifies whether to print the current timestamp, parent process ID, process ID, and thread ID for all ODBC functions to the output file. The default is `0`.

- If set to `0`, the driver uses standard ODBC tracing.
- If set to `1`, the log file includes a timestamp on ENTRY and EXIT of each ODBC function.
- If set to `2`, the log file prints a header on every line. By default, the header includes the parent process ID and process ID.
- If set to `3`, both `TraceOptions=1` and `TraceOptions=2` are enabled. The header includes a timestamp as well as a parent process ID and process ID.

Example

In the following example of trace settings, tracing has been enabled, the name of the log file is `odbctrace.out`, the library for tracing is `ivtrc28.so`, the maximum size of the log file is 51200 KB, and the maximum number of log files is 8. Timestamp and other information is included in `odbctrace.out`.

```
Trace=1
TraceFile=ODBCHOME/lib/odbctrace.out
TraceDll=ODBCHOME/lib/ivtrc28.so
ODBCTraceMaxFileSize=51200
ODBCTraceMaxNumFiles=8
TraceOptions=3
```

macOS iODBC Administrator

On macOS, you can enable tracing through the Tracing tab of the iODBC Data Source Administrator.

To specify the path and name of the trace log file, type the path and name in the Log file path field or click **Browse** to select a log file. If no location is specified, the trace log resides in the working directory of the application you are using.

The iODBC Data Source Administrator ships with a trace library that is enabled by default. If you want to use a custom library instead, type the path and name of the library in the Custom trace library field or click **Browse** to select the library.

To enable tracing, indicate the frequency of tracing for the "When to trace" option on the Trace tab. If you select **All the time**, tracing continues until you disable it. Be sure to turn off tracing when you are finished reproducing the issue because tracing decreases the performance of your ODBC application.

After making changes on the Tracing tab, click **Apply** for them to take effect.

The DataDirect enhanced tracing library gives you more control over tracing. See "System Information (odbc.ini) File" for a complete discussion of how to configure enhanced tracing.

See also

System information (odbc.ini) file on page 15

Test loading tool

Before using the test loading tool, be sure that your environment variables are set correctly. Refer to "Environment variable" in the user's guide for your driver for details.

The ivtestlib (32-bit drivers) and ddtestlib (64-bit drivers) test loading tools are provided to test load drivers and help diagnose configuration problems in the UNIX, Linux, and macOS environments, such as environment variables not correctly set or missing database client components. This tool is installed in the `/bin` subdirectory in the product installation directory. It attempts to load a specified ODBC driver and prints out all available error information if the load fails.

For example, if the drivers are installed in `/opt/odbc/lib`, the following command attempts to load the 32-bit driver on Solaris, where *xx* represents the version number of the driver:

```
ivtestlib /opt/odbc/lib/ivoraxx.so
```

Note: On the HP-UX version, the full path to the driver must be specified for the tool. For other platforms, the full path is not required.

If the load is successful, the tool returns a success message along with the version string of the driver. If the driver cannot be loaded, the tool returns an error message explaining why.

For version string details, refer to "Version string information" in the user's guide for your driver.

ODBC Test

On Windows, Microsoft® ships with its ODBC SDK an ODBC-enabled application, named ODBC Test, that you can use to test ODBC drivers and the ODBC Driver Manager. ODBC 3.52 includes both ANSI and Unicode-enabled versions of ODBC Test.

To use ODBC Test, you must understand the ODBC API, the C language, and SQL. For more information about ODBC Test, refer to the *Microsoft ODBC SDK Guide*.

iODBC Demo and iODBC Test

On macOS, the iODBC Driver Manager includes two sample applications, iODBC Demo and iODBC Test, that you can use to test ODBC drivers and the ODBC Driver Manager. iODBC Demo supports a graphical user interface to run tests, while iODBC Test employs a command-line interface. Both applications allow you to execute SQL statements against your environment, providing a quick means to test your connections, configurations, and setup. ANSI and Unicode-enabled versions of both applications are installed with the Driver Manager.

Logging for Java components

The following Progress DataDirect drivers for ODBC include a flexible and comprehensive logging mechanism for internal Java components.

- Aha!
- Apache Cassandra
- Autonomous REST Connector
- GitHub
- Google Analytics
- Google BigQuery
- HubSpot
- Microsoft Dynamics 365
- Microsoft SharePoint
- MongoDB
- Oracle Service Cloud
- Salesforce
- SAP S/4HANA
- TeamCity

This logging mechanism allows logging to be incorporated seamlessly with the logging of your application or enabled and configured independently of the application. The logging mechanism can be instrumental in investigating and diagnosing issues. It also provides valuable insight into the type and number of operations requested by the application from the driver and requested by the driver from the data source. This information can help you tune and optimize your application.

Loggers and logging levels

The Java Logging API is used to configure and control the loggers used by the driver. The Java Logging API is built into the JVM.

The Java Logging API allows applications or components to define one or more named loggers. Messages written to the loggers can be given different levels of importance. For example, warnings that occur in the driver can be written to a logger at the WARNING level, while progress or flow information can be written to a logger at the INFO or FINER level. Each logger used by the driver can be configured independently. The configuration for a logger includes what level of log messages are written, the location to which they are written, and the format of the log message.

The Java Logging API defines the following levels:

- SEVERE
- CONFIG
- FINE
- FINER
- FINEST
- INFO
- WARNING

Note: Log messages logged by the driver only use the CONFIG, FINE, FINER, and FINEST logging levels.

Setting the log threshold of a logger to a particular level causes the logger to write log messages of that level and higher to the log. For example, if the threshold is set to `FINE`, the logger writes messages of levels FINE, CONFIG, and SEVERE to its log. Messages of level FINER or FINEST are not written to the log.

The driver exposes loggers for the following functional areas:

- Driver to SQL Communication
- SQL Engine
- Web service adapter

Driver to SQL communication logger

Name

datadirect.cloud.drivercommunication

Description

Logs all calls made by the driver to the SQL Engine and the responses from the SQL Engine back to the driver.

Message Levels

`CONFIG` - Errors and Warnings encountered by the communication protocol are logged at this level.

`FINER` - The message type and arguments for requests and responses sent between the driver and SQL Engine are logged at this level. Data transferred between the driver and SQL Engine is not logged.

`FINEST` - Data transferred between the driver and SQL Engine is logged at this level.

Default

OFF

SQL engine logger

Name

datadirect.cloud.sql.level

Description

Logs the operations that the SQL engine performs while executing a query. Operations include preparing a statement to be executed, executing the statement, and fetching the data, if needed. These are internal operations that do not necessarily directly correlate with Web service calls made to the remote data source.

Message Levels

`CONFIG` - Any errors or warnings detected by the SQL engine are written at this level.

`FINE` - In addition to the same information logged by the CONFIG level, SQL engine operations are logged at this level. In particular, the SQL statement that is being executed is written at this level.

`FINER` - In addition to the same information logged by the CONFIG and FINE levels, data sent or received in the process of performing an operation is written at this level.

Wire protocol adapter logger

Name

datadirect.cloud.adapter.level

Description

Logs the calls the driver makes to the remote data source and the responses it receives from the remote data source.

Message Levels

`CONFIG` - Any errors or warnings detected by the wire protocol adapter are written at this level.

`FINE` - In addition to the information logged by the CONFIG level, information about calls made by the wire protocol adapter and responses received by the wire protocol adapter are written at this level. In particular, the calls made to execute the query and the calls to fetch or send the data are logged. The log entries for the calls to execute the query include the API-specific query being executed. The actual data sent or fetched is not written at this level.

`FINER` - In addition to the information logged by the CONFIG and FINE levels, this level provides additional information.

`FINEST` - In addition to the information logged by the CONFIG, FINE, and FINER levels, data associated with the calls made by the wire protocol adapter is written.

Configuring logging

You can configure logging using a standard Java properties file in either of the following ways:

- Using the properties file that is shipped with your JVM. See "Using the JVM" for details.
- Using the driver. See "Using the driver" for details.

Using the JVM

If you want to configure logging using the properties file that is shipped with your JVM, use a text editor to modify the properties file in your JVM. Typically, this file is named `logging.properties` and is located in the `JRE/lib` subdirectory of your JVM. The JRE looks for this file when it is loading.

You can also specify which properties file to use by setting the `java.util.logging.config.file` system property. At a command prompt, enter:

```
java -Djava.util.logging.config.file=properties_file
```

where *`properties_file`* is the name of the properties file you want to load.

Using the driver

If you want to configure logging using the driver, you can use either of the following approaches:

- Use a single properties file for all connections.
- Use a different properties file for each schema map. For example, if you have two schema maps (`C:\data\schemamaps1\` and `C:\data\schemamaps2\`, for example), you can load one properties file for the `test1map.config` schema map and load another properties file for the `test2map.config` schema map.

By default, the driver looks for the file named `ddlogging.properties` in the current working directory to load for all connections. If the driver is operating in Server mode, the driver uses the `ddlogging.properties` file that is stored in the application working directory.

If a properties file is specified for the LogConfigFile connection option, the driver uses the following process to determine which file to load:

1. The driver looks for the file specified by the LogConfigFile connection option.
2. If the driver cannot find the file in Step 1 on page 21, it looks for a properties file named *`user_name`*`.logging.properties` in the application working directory, where *`user_name`* is your user ID used to connect to the REST service.
3. If the driver cannot find the file in Step 2 on page 21, it looks for a properties file named `ddlog.properties` in the current working directory.
4. If the driver cannot find the file in Step 3 on page 21, it abandons its attempt to load a properties file.

If any of these files exist, but the logging initialization fails for some reason while using that file, the driver writes a warning to the standard output (System.out), specifying the name of the properties file being used.

A sample properties filenamed `ddlogging.properties` is installed in the *`install_dir\samples`* subdirectory of your product installation directory, where *`install_dir`* is your product installation directory.

The demoodbc Application

DataDirect provides a simple C application, named demoodbc, that is useful for:

- Executing `SELECT * FROM emp`, where `emp` is a database table. The scripts for building the `emp` database tables (one for each supported database) are in the demo subdirectory in the product installation directory.
- Testing database connections.
- Creating reproducibles.
- Persisting data to an XML data file.

The demoodbc application is installed in the `/samples/demo` subdirectory in the product installation directory. Refer to demoodbc.txt or demoodbc64.txt in the demo directory for an explanation of how to build and use this application.

The example application

Progress DataDirect provides a simple C application, named example, that is useful for:

- Executing any type of SQL statement
- Testing database connections
- Testing SQL statements
- Verifying your database environment

The example application is installed in the `/samples/example` subdirectory in the product installation directory. Refer to `example.txt` or `example64.txt` in the example directory for an explanation of how to build and use this application.

Enabling debug record mode

Supported by the following drivers:

- Aha!
- Autonomous REST Connector
- GitHub
- Google Analytics
- HubSpot
- TeamCity

The drivers supports a debug record mode that provides a method for troubleshooting issues that occur when accessing data on a REST service. When Debug Record Mode is enabled, the driver captures and records server requests and responses to a set of files stored in a designated location. Technical Support can then use these files to analyze and reproduce the issue without requiring access to your private data source.

To generate debug record files using the Configuration Manager dialog:

1. On the Diagnostics tab, using the Debug Folder field, specify the location where you want the driver to generate the files used to record server requests and responses.
2. Start the ODBC application and reproduce the issue.
3. Stop the application.

To generate debug record files using the Setup dialog (Windows):

1. On the Advanced tab, check the **Record** box in the Debug Record section.
2. Click the **Select...** button associated with the Debug Folder field. The **Browse For Folder** dialog opens.
3. From the **Browse For Folder** dialog, select or create the location you want the driver to generate the files used to record server requests and responses; then, click **OK**.

4. Start the ODBC application and reproduce the issue.
5. Stop the application.

To generate debug record files (non-GUI):

1. Using the DebugRecord connection option, specify the location where you want the driver to generate the files used to record server requests and responses.
2. Start the ODBC application and reproduce the issue.
3. Stop the application.

Results: The driver generates a set of files containing the server requests and responses that occurred during the session.

Contact Technical Support for assistance analyzing the files and reproducing the issue.

Important: Debug record files may capture security-related headers, such as auth or token headers. Before sending Technical Support debug files, review the content to remove any confidential information that may have been recorded.

What to do next: After generating the debug files, you can remove the location specified for the Debug Record/Debug Folder (DebugRecord) option or, if using the Setup dialog, deselect the **Record** check box. If you do not remove this value, the driver will overwrite debug files in the specified location the next time you start the application.

Other tools

The Progress DataDirect Support Web site provides other diagnostic tools that you can download to assist you with troubleshooting. These tools are not shipped with the product. Refer to the Progress DataDirect Web page:

https://www.progress.com/support/evaluation/download-resources/download-tools

Progress DataDirect also provides a knowledgebase that is useful in troubleshooting problems. Refer to the Progress DataDirect Knowledgebase page:

http://progresscustomersupport-survey.force.com/ConnectKB

Error messages

Error messages can be generated from:

- ODBC driver
- Database system
- ODBC driver manager

An error reported on an ODBC driver has the following format:

```
[vendor] [ODBC_component] message
```

where *ODBC_component* is the component in which the error occurred. For example, an error message from the Progress DataDirect *for* ODBC for Oracle Wire Protocol driver would look like this:

```
[DataDirect] [ODBC Oracle Wire Protocol Driver] Invalid precision specified.
```

If you receive this type of error, check the last ODBC call made by your application for possible problems or contact your ODBC application vendor.

An error that occurs in the data source includes the data store name, in the following format:

```
[vendor] [ODBC_component] [data_store] message
```

With this type of message, *ODBC_component* is the component that received the error specified by the data store. For example, you may receive the following message from an Oracle database:

```
[DataDirect] [ODBC Oracle Wire Protocol Driver] [Oracle] ORA-0919: specified length too
 long for CHAR column
```

This type of error is generated by the database system. Check your database system documentation for more information or consult your database administrator.

On Windows, the Microsoft Driver Manager is a DLL that establishes connections with drivers, submits requests to drivers, and returns results to applications. An error that occurs in the Driver Manager has the following format:

```
[vendor] [ODBC XXX] message
```

For example, an error from the Microsoft Driver Manager might look like this:

```
[Microsoft] [ODBC Driver Manager] Driver does not support this function
```

If you receive this type of error, consult the *Programmer's Reference* for the Microsoft ODBC Software Development Kit available from Microsoft.

UNIX®

On UNIX and Linux, the Driver Manager is provided by Progress DataDirect. For example, an error from the DataDirect Driver Manager might look like this:

```
[DataDirect][ODBC lib] String data code page conversion failed.
```

UNIX, Linux, and macOS error handling follows the X/Open XPG3 messaging catalog system. Localized error messages are stored in the subdirectory:

```
locale/localized_territory_directory/LC_MESSAGES
```

where *localized_territory_directory* depends on your language.

For instance, German localization files are stored in `locale/de/LC_MESSAGES`, where `de` is the locale for German.

If localized error messages are not available for your locale, then they will contain message numbers instead of text. For example:

```
[DataDirect] [ODBC 20101 driver] 30040
```

On macOS, the iODBC Driver Manager establishes connections with drivers, submits requests to drivers, and returns results to applications. An error that occurs in the Driver Manager has the following format:

```
[vendor] [Driver Manager] message
```

For example, an error from the Microsoft Driver Manager might look like this:

```
[iODBC] [Driver Manager] Specified driver could not be loaded
```

If you receive this type of error, consult the iODBC documentation at http://www.iodbc.org/.

UNIX, Linux, and macOS error handling follows the X/Open XPG3 messaging catalog system. Localized error messages are stored in the subdirectory:

```
locale/localized_territory_directory/LC_MESSAGES
```

where *localized_territory_directory* depends on your language.

For instance, German localization files are stored in `locale/de/LC_MESSAGES`, where `de` is the locale for German.

If localized error messages are not available for your locale, then they will contain message numbers instead of text. For example:

```
[DataDirect] [ODBC 20101 driver] 30040
```

Troubleshooting issues

If you are having an issue while using your driver, first determine the type of issue that you are encountering:

- Setup/connection
- Interoperability (ODBC application, ODBC driver, ODBC Driver Manager, or data source)
- Performance

This chapter describes these three types of issues, provides some typical causes of the issues, lists some diagnostic tools that are useful to troubleshoot the issues, and, in some cases, explains possible actions you can take to resolve the issues.

Setup/connection issues

You are experiencing a setup/connection issue if you are encountering an error or hang while you are trying to make a database connection with the ODBC driver or are trying to configure the ODBC driver.

Some common errors that are returned by the ODBC driver if you are experiencing a setup/connection issue include:

- `Specified driver could not be loaded.`
- `Data source name not found and no default driver specified.`

- `Cannot open shared library: libodbc.so.`
- `Unable to connect to destination.`
- `Invalid username/password; logon denied.`

Troubleshooting the issue

Some common reasons that setup/connection issues occur are:

- On Windows, UNIX, and Linux, the library path environment variable is not set correctly.

  HP-UX ONLY:

  - When setting the library path environment variable on HP-UX operating systems, specifying the parent directory is not required.
  - You also must set the `LD_PRELOAD` environment variable to the fully qualified path of the `libjvm.so`[`sl`].

  The library path environment variable is:

  32-bit Drivers

  - `PATH` on Windows
  - `LD_LIBRARY_PATH` on Solaris, Linux and HP-UX Itanium
  - `SHLIB_PATH` on HP-UX PA_RISC
  - `LIBPATH` on AIX

  64-bit Drivers

  - `PATH` on Windows
  - `LD_LIBRARY_PATH` on Solaris, HP-UX Itanium, and Linux
  - `LIBPATH` on AIX

- The database and/or listener are not started.
- The `ODBCINI` environment variable is not set correctly for the ODBC drivers on UNIX, Linux, or macOS.
- The ODBC driver's connection attributes are not set correctly in the system information file on UNIX, Linux, and macOS. For more information, refer to "Data Source Configuration on UNIX/Linux" or "Data Source Configuration for macOS" in your driver's user's guide. For example, the host name or port number are not correctly configured. Refer to "Connection Option Descriptions" in your driver's user's guide for a list of connection string attributes that are required for each driver to connect properly to the underlying database.

See "The Test Loading Tool" for information about a helpful diagnostic tool.

See also

Test loading tool on page 17

Interoperability issues

Interoperability issues can occur with a working ODBC application in any of the following ODBC components: ODBC application, ODBC driver, ODBC Driver Manager, and/or data source.

For example, any of the following problems may occur because of an interoperability issue:

- SQL statements may fail to execute.
- Data may be returned/updated/deleted/inserted incorrectly.
- A hang or core dump may occur.

Troubleshooting the issue

Isolate the component in which the issue is occurring. Is it an ODBC application, an ODBC driver, an ODBC Driver Manager, or a data source issue?

To troubleshoot the issue:

1. Test to see if your ODBC application is the source of the problem. To do this, replace your working ODBC application with a more simple application. If you can reproduce the issue, you know your ODBC application is **not** the cause.

   UNIX®

   On UNIX and Linux, you can use the example application that is shipped with your driver. See "The example Application" for details.

   On Windows, you can use ODBC Test, which is part of the Microsoft ODBC SDK, or the example application that is shipped with your driver. See "ODBC Test" and "The example Application" for details.

   Mac OS

   On macOS, you can use iODBC Demo or iODBC Test, which are installed with the iODBC Administrator, or the example application that is shipped with your driver. See "iODBC Demo and iODBC Test" and "The example Application" for details.

2. Test to see if the data source is the source of the problem. To do this, use the native database tools that are provided by your database vendor.
3. If neither the ODBC application nor the data source is the source of your problem, troubleshoot the ODBC driver and the ODBC Driver Manager.

   In this case, we recommend that you create an ODBC trace log to provide to Technical Support. See "ODBC Trace" for details.

See also

ODBC Test on page 17
The example application on page 22
iODBC Demo and iODBC Test on page 18
ODBC trace on page 13

Performance issues

Developing performance-oriented ODBC applications is not an easy task. You must be willing to change your application and test it to see if your changes helped performance. Microsoft's *ODBC Programmer's Reference* does not provide information about system performance. In addition, ODBC drivers and the ODBC Driver Manager do not return warnings when applications run inefficiently.

Some general guidelines for developing performance-oriented ODBC applications include:

- Use catalog functions appropriately.
- Retrieve only required data.
- Select functions that optimize performance.
- Manage connections and updates.

See "Designing ODBC applications for performance optimization" for complete information.

See also

Designing ODBC applications for performance optimization on page 73

3

Failover

A number of Progress DataDirect drivers support failover to ensure continuous, uninterrupted access to data.

Note: For implementation and configuration details, refer to the "Using failover" topic in the user's guide for your driver.

Note: While not all drivers support failover, most drivers include the ConnectionRetryCount and ConnectionRetryDelay connection properties to support continuous access to data.

The the following levels of failover protection (listed from basic to more comprehensive) may be available with your driver:

- *Connection failover* provides failover protection for new connections only. The driver fails over new connections to an alternate, or backup, database server if the primary database server is unavailable, for example, because of a hardware failure or traffic overload. If a connection to the database is lost, or dropped, the driver does not fail over the connection. This failover method is the default.
- *Extended connection failover* provides failover protection for new connections and lost database connections. If a connection to the database is lost, the driver fails over the connection to an alternate server, preserving the state of the connection at the time it was lost, but not any work in progress.
- *Select Connection failover* provides failover protection for new connections and lost database connections. In addition, it provides protection for Select statements that have work in progress. If a connection to the database is lost, the driver fails over the connection to an alternate server, preserving the state of the connection at the time it was lost and preserving the state of any work being performed by Select statements.

The method you choose depends on how failure tolerant your application is. For example, if a communication failure occurs while processing, can your application handle the recovery of transactions and restart them? Your application needs the ability to recover and restart transactions when using either extended connection failover mode or select connection failover mode. The advantage of select mode is that it preserves the state of any work that was being performed by the Select statement at the time of connection loss. If your application had been iterating through results at the time of the failure, when the connection is reestablished the driver can reposition on the same row where it stopped so that the application does not have to undo all of its previous result processing. For example, if your application were paging through a list of items on a Web page when a failover occurred, the next page operation would be seamless instead of starting from the beginning. Performance, however, is a factor in selecting a failover mode. Select mode incurs additional overhead when tracking what rows the application has already processed.

You can specify which failover method you want to use by setting the Failover Mode connection option. Regardless of the failover method you choose, you must configure one or multiple alternate servers using the Alternate Servers connection option.

For details, see the following topics:

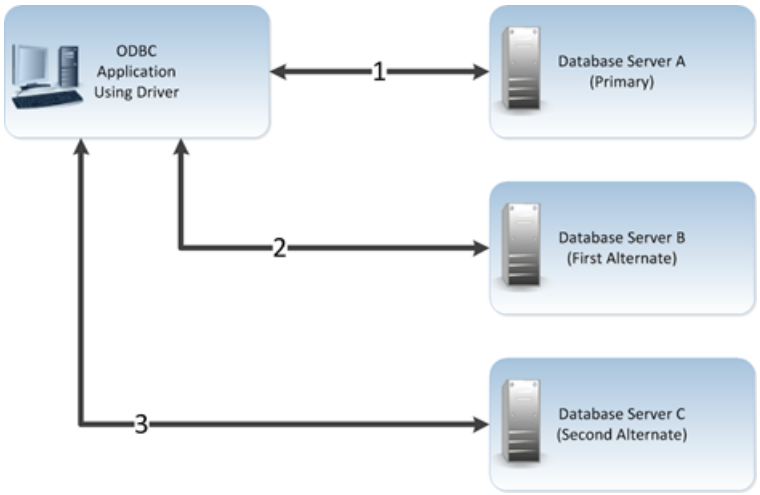
- Connection failover
- Extended connection failover
- Select connection failover
- Guidelines for primary and alternate servers
- Using client load balancing
- Using connection retry
- Summary of failover-related options

Connection failover

Connection failover allows an application to connect to an alternate, or backup, database server if the primary database server is unavailable, for example, because of a hardware failure or traffic overload. Connection failover provides failover protection for new connections only and does not provide protection for lost connections to the database, nor does it preserve states for transactions or queries.

You can customize the drivers for connection failover by configuring a list of alternate database servers that are tried if the primary server is not accepting connections. Connection attempts continue until a connection is successfully established or until all the alternate database servers have been tried the specified number of times.

For example, suppose you have the environment shown in the following illustration with multiple database servers: Database Server A, B, and C. Database Server A is designated as the primary database server, Database Server B is the first alternate server, and Database Server C is the second alternate server.



First, the application attempts to connect to the primary database server, Database Server A (**1**). If connection failover is enabled and Database Server A fails to accept the connection, the application attempts to connect to Database Server B (**2**). If that connection attempt also fails, the application attempts to connect to Database Server C (**3**).

In this scenario, it is probable that at least one connection attempt would succeed, but if no connection attempt succeeds, the driver can retry each alternate database server (primary and alternate) for a specified number of attempts. You can specify the number of attempts that are made through the *connection retry* feature. You can also specify the number of seconds of delay, if any, between attempts through the *connection delay* feature. See Using connection retry on page 34 for more information about connection retry.

A driver fails over to the next alternate database server only if a successful connection cannot be established with the current alternate server. If the driver successfully establishes communication with a database server and the connection request is rejected by the database server because, for example, the login information is invalid, then the driver generates an error and does not try to connect to the next database server in the list. It is assumed that each alternate server is a mirror of the primary and that all authentication parameters and other related information are the same.

Extended connection failover

Extended connection failover provides failover protection for the following types of connections:

- New connections, in the same way as described in Connection failover on page 30
- Lost connections

When a connection to the database is lost, the driver fails over the connection to an alternate server, restoring the same state of the connection at the time it was lost. For example, when reestablishing a lost connection on the alternate database server, the driver performs the following actions:

- Restores the connection using the same connection options specified by the lost connection
- Reallocates statement handles and attributes
- Logs in the user to the database with the same user credentials
- Restores any prepared statements associated with the connection and repopulates the statement pool
- Restores manual commit mode if the connection was in manual commit mode at the time of the failover

The driver does not preserve work in progress. For example, if the database server experienced a hardware failure while processing a query, partial rows processed by the database and returned to the client would be lost. If the driver was in manual commit mode and one or more Inserts or Updates were performed in the current transaction before the failover occurred, then the transaction on the primary server is rolled back. The Inserts or Updates done before the failover are not committed to the primary server. Your application needs to rerun the transaction after the failover because the Inserts or Updates done before the failover are not repeated by the driver on the failover connection.

When a failover occurs, if a statement is in allocated or prepared state, the next operation on the statement returns a SQL state of 01000 and a vendor code of 0. If a statement is in an executed or prepared state, the next operation returns a SQL state of 40001 and a vendor code of 0. Either condition returns an error message similar to:

```
Your connection has been terminated. However, you have been successfully connected to
the next available AlternateServer: 'HOSTNAME=Server4:PORTNUMBER= 1521:SERVICENAME=test'.
 All active transactions have been rolled back.
```

The driver retains all connection settings made through ODBC API calls when a failover connection is made. It does not, however, retain any session settings established through SQL statements. This can be done through the Initialization String connection option, described in the individual driver chapters.

The driver retains the contents of parameter buffers, which can be important when failing over after a fetch. All Select statements are re-prepared at the time the failover connection is made. All other statements are placed in an allocated state.

If an error occurs while the driver is reestablishing a lost connection, the driver can fail the entire failover process or proceed with the process as far as it can. For example, suppose an error occurred while reestablishing the connection because a table for which the driver had a prepared statement did not exist on the alternate connection. In this case, you may want the driver to notify your application of the error and proceed with the failover process. You can choose how you want the driver to behave if errors occur during failover by setting the Failover Granularity connection option.

During the failover process, your application may experience a short pause while the driver establishes a connection on an alternate server. If your application is time-sensitive (a real-time customer order application, for example) and cannot absorb this wait, you can set the Failover Preconnect connection option to true. Setting the Failover Preconnect option to true instructs the driver to establish connections to the primary server and an alternate server at the same time. Your application uses the first connection that is successfully established. If this connection to the database is lost at a later time, the driver saves time in reestablishing the connection on the server to which it fails over because it can use the spare connection in its failover process.

This pre-established failover connection is not used by the driver until the driver determines that it needs to fail over. If the server to which the driver is connected or the network equipment through which the connection is routed is configured with a timeout, the pre-configured failover connection could time out. The pre-configured failover connection can also be lost if the failover server is brought down and back up again. The driver tries to establish the connection to the failover server again if the connection is lost.

Select connection failover

Select connection failover provides failover protection for the following types of connections:

- New connections, in the same way as described in Connection failover on page 30
- Lost connections, in the same way as described in Extended connection failover on page 31

In addition, the driver can recover work in progress because it keeps track of the last Select statement the application executed on each Statement handle, including how many rows were fetched to the client. For example, if the database had only processed 500 of 1,000 rows requested by a Select statement when the connection was lost, the driver would reestablish the connection to an alternate server, re-execute the Select statement, and position the cursor on the next row so that the driver can continue fetching the balance of rows as if nothing had happened.

Performance, however, is a factor when considering whether to use Select mode. Select mode incurs additional overhead when tracking what rows the application has already processed.

The driver only recovers work requested by Select statements. You must explicitly restart the following types of statements after a failover occurs:

- Insert, Update, or Delete statements
- Statements that modify the connection state, for example, SET or ALTER SESSION statements
- Objects stored in a temporary tablespace or global temporary table
- Partially executed stored procedures and batch statements

When in manual transaction mode, no statements are rerun if any of the operations in the transaction were Insert, Update, or Delete. This is true even if the statement in process at the time of failover was a Select statement.

By default, the driver verifies that the rows that are restored match the rows that were originally fetched and, if they do not match, generates an error warning your application that the Select statement must be reissued. By setting the Failover Granularity connection option, you can customize the driver to ignore this check altogether or fail the entire failover process if the rows do not match.

When the row comparison does not agree, the default behavior of Failover Granularity returns a SQL state of 40003 and an error message similar to:

```
Unable to position to the correct row after a successful failover attempt to
AlternateServer: 'HOSTNAME=Server4:PORTNUMBER= 1521:SERVICENAME=test'. You must reissue
 the select statement.
```

If you have configured Failover Granularity to fail the entire failover process, the driver returns a SQL state of 08S01 and an error message similar to:

```
Your connection has been terminated and attempts to complete the failover process to the
 following Alternate Servers have failed: AlternateServer: 'HOSTNAME=Server4:PORTNUMBER=
 1521:SERVICENAME=test'. All active transactions have been rolled back.
```

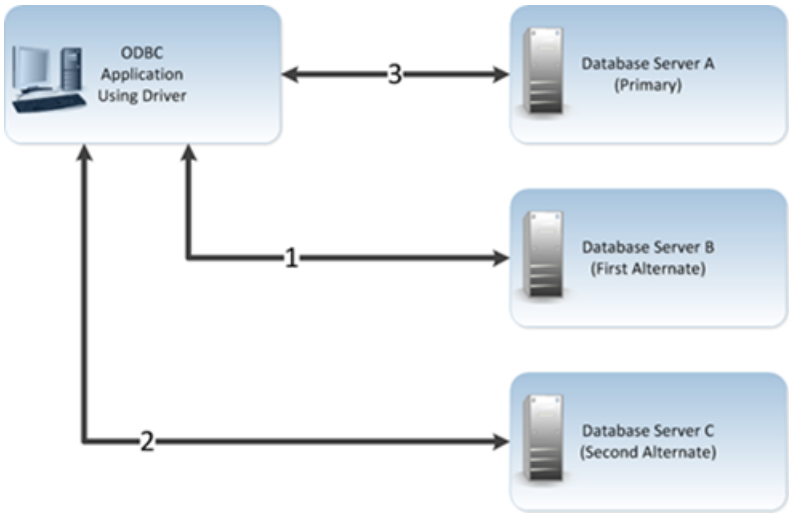
Guidelines for primary and alternate servers

Many databases provide advanced database replication technologies such as DB2 High Availability Disaster Recovery (HADR) and Oracle Real Application Clusters (RAC), and Microsoft Cluster Server (MSCS). The failover functionality provided by the drivers does not require any of these technologies, but can work with them to provide comprehensive failover protection. Use the following guidelines for primary and alternate servers to ensure that failover works correctly in your environment:

- Alternate servers should mirror data on the primary server or be part of a configuration where multiple database nodes share the same physical data.
- If using failover with DB2 HADR, the primary server must be the primary server configured in your HADR system and any alternate server must be a standby server configured in your HADR system.

Using client load balancing

Client load balancing helps distribute new connections in your environment so that no one server is overwhelmed with connection requests. When client load balancing is enabled, the order in which primary and alternate database servers are tried is random. For example, suppose that client load balancing is enabled as shown in the following illustration:



First, Database Server B is tried (**1**). Then, Database Server C may be tried (**2**), followed by a connection attempt to Database Server A (**3**). In contrast, if client load balancing were not enabled in this scenario, each database server would be tried in sequential order, primary server first, then each alternate server based on its entry order in the alternate servers list.

Client load balancing is controlled by the Load Balancing connection option.

Using connection retry

Connection retry defines the number of times the driver attempts to connect to the primary server and, if configured, alternate database servers after the initial unsuccessful connection attempt. It can be used with connection failover, extended connection failover, and select failover. Connection retry can be an important strategy for system recovery. For example, suppose you have a power failure in which both the client and the server fails. When the power is restored and all computers are restarted, the client may be ready to attempt a connection before the server has completed its startup routines. If connection retry is enabled, the client application can continue to retry the connection until a connection is successfully accepted by the server.

Connection retry can be used in environments that have only one server or can be used as a complementary feature with connection failover in environments with multiple servers.

Using the connection options Connection Retry Count and Connection Retry Delay, you can specify the number of times the driver attempts to connect and the time in seconds between connection attempts.

Summary of failover-related options

The following table summarizes how failover-related connection options work with the drivers. See "Connection Option Descriptions" in the user's guide for your driver for further details. Not all options are available in every failover-enabled driver. The step numbers in the table refer the procedure that follows the table

Table 1: Summary: Failover and Related Connection Options

Option	Characteristic
Alternate Servers (See step 1 on page 35)	One or multiple alternate database servers. An IP address or server name identifying each server is required.
Connection Retry Count (See step 5 on page 36)	Number of times the driver retries the primary database server, and if specified, alternate servers until a successful connection is established.
Connection Retry Delay (See step 6 on page 36)	Wait interval, in seconds, between connection retry attempts when the Connection Retry Count option is set to a positive integer.
Failover Granularity (See step 3 on page 35)	The type of behavior that the driver exhibits when errors are detected during the failover process.
Failover Mode (See step 2 on page 35)	The type of failover that the driver attempts.
Failover Preconnect (See step 4 on page 36)	Determines whether the driver makes a connection attempt to the next server in the Alternate Servers list at the time of the initial connection.
Load Balancing (See step 7 on page 36)	Determines whether the driver uses client load balancing in its attempts to connect to primary and alternate database servers. If enabled, the driver attempts to connect to the database servers in random order.

1. To configure connection failover, you **must** specify one or more alternate database servers that are tried at connection time if the primary server is not accepting connections. To do this, use the Alternate Servers connection option. Connection attempts continue until a connection is successfully established or until all the database servers in the list have been tried once (the default).
2. Choose a failover method by setting the Failover Mode connection option. The default method is Connection (FailoverMode=0).
3. If Failover Mode is Extended Connection (FailoverMode=1) or Select (FailoverMode=2), set the Failover Granularity connection option to specify how you want the driver to behave if errors occur while trying to reestablish a lost connection. The default behavior of the driver is Non-Atomic (FailoverGranularity=0), which continues with the failover process and posts any errors on the statement on which they occur. Other values are:

   Atomic (FailoverGranularity=1): the driver fails the entire failover process if an error is generated as the result of anything other than executing and repositioning a Select statement. If an error is generated as a result of repositioning a result set to the last row position, the driver continues with the failover process, but generates a warning that the Select statement must be reissued.

Atomic including Repositioning (FailoverGranularity=2): the driver fails the entire failover process if any error is generated as the result of restoring the state of the connection or the state of work in progress.

Disable Integrity Check (FailoverGranularity=3: the driver does not verify that the rows restored during the failover process match the original rows. This value applies only when Failover Mode is set to Select (FailoverMode=2).

4. Optionally, enable the Failover Preconnect connection option (FailoverPreconnect=1) if you want the driver to establish a connection with the primary and an alternate server at the same time. This value applies only when Failover Mode is set to Extended Connection (FailoverMode=1) or Select (FailoverMode=2). The default behavior is to connect to an alternate server only when failover is caused by an unsuccessful connection attempt or a lost connection (FailoverPreconnect=0).
5. Optionally, specify the number of times the driver attempts to connect to the primary and alternate database servers after the initial unsuccessful connection attempt. By default, the driver does not retry. To set this feature, use the Connection Retry Count connection option.
6. Optionally, specify the wait interval, in seconds, between attempts to connect to the primary and alternate database servers. The default interval is 3 seconds. To set this feature, use the Connection Retry Delay connection option.
7. Optionally, specify whether the driver will use client load balancing in its attempts to connect to primary and alternate database servers. If load balancing is enabled, the driver uses a random pattern instead of a sequential pattern in its attempts to connect. The default value is not to use load balancing. To set this feature, use the Load Balancing connection option.

A connection string example

The following connection string configures the Oracle Wire Protocol driver to use connection failover in conjunction with some of its optional features.

```
DSN=AcctOracleServer;AlternateServers=(HostName=AccountingOracleServer:PortNumber=1521:
SID=Accounting, HostName=255.201.11.24:PortNumber=1522:ServiceName=ABackup.NA.MyCompany);
ConnectionRetryCount=4;ConnectionRetryDelay=5;LoadBalancing=1;FailoverMode=0
```

Specifically, this connection string configures the driver to use two alternate servers as connection failover servers, to attempt to connect four additional times if the initial attempt fails, to wait five seconds between attempts, to try the primary and alternate servers in a random order, and to attempt reconnecting on new connections only. The additional connection information required for the alternate servers is specified in the data source AcctOracleServer.

An odbc.ini file example

To configure the 32-bit Oracle Wire Protocol driver to use connection failover in conjunction with some of its optional features in your odbc.ini file, you could set the following connection string attributes:

```
Driver=ODBCHOME/lib/ivoraXX.so
Description=DataDirect Oracle Wire Protocol driver
...
AlternateServers=(HostName=AccountingOracleServer:PortNumber=1521:SID=Accounting,
HostName=255.201.11.24:PortNumber=1522:ServiceName=ABackup.NA.MyCompany)
...
ConnectionRetryCount=4
ConnectionRetryDelay=5
...
LoadBalancing=0
...
FailoverMode=1
...
FailoverPreconnect=1
...
```

Specifically, this odbc.ini configuration tells the driver to use two alternate servers as connection failover servers, to attempt to connect four additional times if the initial attempt fails, to wait five seconds between attempts, to try the primary and alternate servers in sequential order (do not use load balancing), to attempt reconnecting on new and lost connections, and to establish a connection with the primary and alternate servers at the same time.

4

Client information

Many databases allow applications to store client information associated with a connection. For example, the following types of information can be useful for database administration and monitoring purposes:

- Name of the application currently using the connection.
- User ID for whom the application using the connection is performing work. The user ID may be different than the user ID that was used to establish the connection.
- Host name of the client on which the application using the connection is running.
- Product name and version of the driver on the client.
- Additional information that may be used for accounting or troubleshooting purposes, such as an accounting ID.

Client information is available in the following ODBC drivers:

- DB2 Wire Protocol
- Oracle Wire Protocol

For DB2 V9.5 and higher for Linux/UNIX/Windows and DB2 for z/OS, this information can feed directly into the Workload Manager (WLM) for workload management and monitoring purposes.

For Oracle 11*g* R2 and higher, this information is managed through the client information feature.

For details, see the following topics:

- How databases store client information
- Storing client information

How databases store client information

Typically, databases that support storing client information do so by providing a register, a variable, or a column in a system table in which the information is stored. If an application attempts to store information and the database does not provide a mechanism for storing that information, the driver caches the information locally. Similarly, if an application returns client information and the database does not provide a mechanism for storing that information, the driver returns the locally cached value.

Storing client information

Your application can store client information associated with a connection. The following table shows the driver connection options that your application can use to store client information and where that client information is stored for each database. See the specific driver chapters for a description of each option.

Table 2: Database Locations for Storing Client Information

Option	Description	Database	Location
Accounting Info	Additional information that may be used for accounting or troubleshooting purposes, such as an accounting ID	DB2	CURRENT CLIENT_ACCTNG register (DB2 for Linux/UNIX/Windows) or CLIENT ACCTNG register (DB2 for z/OS).
		Oracle	CLIENT_INFO value in the V$SESSION table.
Action	The current action within the current module.	Oracle	ACTION value in the V$SESSION table.
Application Name	Name of the application currently using the connection	DB2	CURRENT CLIENT_APPLNAME register (DB2 for Linux/UNIX/Windows) or CLIENT APPLNAME register (DB2 for z/OS). For DB2 V9.1 and higher for Linux/UNIX/Windows, this value is also stored in the APPL_NAME value in the SYSIBMADM.APPLICATIONS table.
		Oracle	CLIENT_IDENTIFIER attribute. In addition, this value is also stored in the PROGRAM value in the V$SESSION table.
Client Host Name	Host name of the client on which the application using the connection is running	DB2	CURRENT CLIENT_WRKSTNNAME register (DB2 for Linux/UNIX/Windows) or CLIENT WRKSTNNAME register (DB2 for z/OS).
		Oracle	MACHINE value in the V$SESSION table.
Client ID	Additional information about the client	Oracle	CLIENT_IDENTIFIER value in the V$SESSION table.

Option	Description	Database	Location
Client User	User ID for whom the application using the connection is performing work	DB2	CURRENT CLIENT_USERID register (DB2 for Linux/UNIX/Windows) or CLIENT USERID register (DB2 for z/OS).
		Oracle	OSUSER value in the V$SESSION table.
Module	The name of a stored procedure or the name of the application	Oracle	MODULE value in the V$SESSION table.
Program ID	Product name and version of the driver on the client	DB2	CLIENT_PRDID value. For DB2 V9.1 and higher for Linux/UNIX/Windows, the CLIENT_PRDID value is located in the SYSIBMADM.APPLICATIONS table.
		Oracle	PROCESS value in the V$SESSION table.

5

Code page values

This chapter lists supported code page values, along with a description, for the Progress DataDirect for ODBC drivers.

For details, see the following topics:

- IANAAppCodePage values
- IBM to IANA code page values
- Teradata code page values

IANAAppCodePage values

The following table lists supported code page values for the IANAAppCodePage connection option. Refer to IANAAppCodePage connection option description in your user's guide for information about this attribute.

To determine the correct numeric value (the MIBenum value) for the IANAAppCodePage connection string attribute, perform the following steps:

1. Determine the code page of your database.
2. Determine the MIBenum value that corresponds to your database code page. To do this, go to:

   http://www.iana.org/assignments/character-sets

On this web page, search for the name of your database code page. This name will be listed as an alias or the name of a character set, and will have a MIBenum value associated with it.

3. Check the following table to make sure that the MIBenum value you looked up on the IANA Web page is supported by the driver. If the value is not listed, contact Progress Technical Support to request support for that value.

Table 3: IANAAppCodePage Values

Value (MIBenum)	Description
3	US_ASCII
4	ISO_8859_1
5	ISO_8859_2
6	ISO_8859_3
7	ISO_8859_4
8	ISO_8859_5
9	ISO_8859_6
10	ISO_8859_7
11	ISO_8859_8
12	ISO_8859_9
16	JIS_Encoding
17	Shift_JIS
18	EUC_JP
30	ISO_646_IRV
36	KS_C_5601
37	ISO_2022_KR
38	EUC_KR
39	ISO_2022_JP
40	ISO_2022_JP_2
57	GB_2312_80
104	ISO_2022_CN
105	ISO_2022_CN_EXT

Value (MIBenum)	Description
109	ISO_8859_13
110	ISO_8859_14
111	ISO_8859_15
113	GBK
2004	HP_ROMAN8
2009	IBM850
2010	IBM852
2011	IBM437
2013	IBM862
2016	IBM-Thai
2024	WINDOWS-31J
2025	GB2312
2026	Big5
2027	MACINTOSH
2028	IBM037
2029	IBM038
2030	IBM273
2033	IBM277
2034	IBM278
2035	IBM280
2037	IBM284
2038	IBM285
2039	IBM290
2040	IBM297
2041	IBM420
2043	IBM424

Value (MIBenum)	Description
2044	IBM500
2045	IBM851
2046	IBM855
2047	IBM857
2048	IBM860
2049	IBM861
2050	IBM863
2051	IBM864
2052	IBM865
2053	IBM868
2054	IBM869
2055	IBM870
2056	IBM871
2062	IBM918
2063	IBM1026
2084	KOI8_R
2085	HZ_GB_2312
2086	IBM866
2087	IBM775
2089	IBM00858
2091	IBM01140
2092	IBM01141
2093	IBM01142
2094	IBM01143
2095	IBM01144
2096	IBM01145

Value (MIBenum)	Description
2097	IBM01146
2098	IBM01147
2099	IBM01148
2100	IBM01149
2102	IBM1047
2250	WINDOWS_1250
2251	WINDOWS_1251
2252	WINDOWS_1252
2253	WINDOWS_1253
2254	WINDOWS_1254
2255	WINDOWS_1255
2256	WINDOWS_1256
2257	WINDOWS_1257
2258	WINDOWS_1258
2259	TIS_620
2000000939 [1]	IBM-939
2000000943 [1]	IBM-943_P14A-2000
2000001025 [1]	IBM-1025
2000004396 [1]	IBM-4396
2000005026 [1]	IBM-5026
2000005035 [1]	IBM-5035

[1] These values are assigned by Progress DataDirect and do not appear in http://www.iana.org/assignments/character-sets.

IBM to IANA code page values

The following table lists the most commonly used IBM code pages and their IANA code page equivalents. These IANA values are valid for the Character Set for CCSID 65535 connection option in the Progress DataDirect DB2 Wire Protocol driver. Refer to the Progress DataDirect DB2 Wire Protocol driver user's guide for more information.

Table 4: IBM to IANA Code Page Values

IBM Number	Value (MIBenum)	IANA Name
37	2028	IBM037
38	2029	IBM038
290	2039	IBM290
300	2000000939[2]	IBM-939
301 [3]	2000000943[2]	IBM-943_P14A-2000
301[4]	2024	WINDOWS-31J
500	2044	IBM500
838	2016	IMB-Thai
857	2047	IBM857
860	2048	IBM860
861	2049	IBM861
897	17	Shift_JIS
913	6	ISO_8859-3
914	7	ISO_8859-4
932	17	Shift_JIS
939	2000000939[2]	IBM-939
943[3]	2000000943[2]	IBM-943_P14A-2000
943[4]	2024	WINDOWS-31J
950	2026	Big5

[2] These values are assigned by Progress DataDirect and do not appear in http://www.iana.org/assignments/character-sets.
[3] If your application runs on a UNIX or Linux platform, use this value.
[4] If your application runs on a Windows platform, use this value.

IBM Number	Value (MIBenum)	IANA Name
1200	1015	UTF-16
1208	106	UTF-8
1250[3]	5	ISO_8859-2
1250[4]	2250	WINDOWS-1250
1251[3]	8	ISO_8859-5
1251[4]	2251	WINDOWS-1251
1252[3]	4	ISO_8859-1
1252[4]	2252	WINDOWS-1252
1253[3]	10	ISO_8859-7
1253[4]	2253	WINDOWS-1253
1254[3]	12	ISO_8859-9
1254[4]	2254	WINDOWS-1254
1255[3]	11	ISO_8859-8
1255[4]	2255	WINDOWS-1255
1256[3]	9	ISO_8859-6
1256[4]	2256	WINDOWS-1256
1257	2257	WINDOWS-1257
1258	2258	WINDOWS-1258
4396	2000004396[2]	IBM-4396
5026	2000005026[2]	IBM-5026
5035	2000005035[2]	IBM-5035
5297	1015	UTF-16
5304	106	UTF-8
13488	1013	UTF-16BE

Teradata code page values

The following table lists code pages that are valid only for the Progress DataDirect Teradata driver. These values do not appear in http://www.iana.org/assignments/character-sets and are assigned by Progress DataDirect. Refer to the Progress DataDirect Teradata driver user's guide for more information.

Table 5: Teradata Code Page Values

Value (MIBenum)	Description
2000005039	ebcdic
2000005040	ebcdic037_0e
2000005041	ebcdic273_0e
2000005042	ebcdic277_0e
2000005043	hangulebcdic933_1ii
2000005044	hangulksc5601_2r4
2000005045	kanjiebcdic5026_0i
2000005046	kanjiebcdic5035_0i
2000005047	kanjieuc_0u
2000005048	kanjisjis_0s
2000005049	katakanaebcdic
2000005050	latin1252_0a
2000005051	latin1_0a
2000005052	latin9_0a
2000005053	schebcdic935_2ij
2000005054	schgb2312_1t0
2000005055	tchbig5_1r0
2000005056	tchebcdic937_3i

6

ODBC API and scalar functions

This chapter lists the ODBC API functions supported by Progress DataDirect for ODBC drivers. In addition, it lists the scalar functions that you use in SQL statements.

For details, see the following topics:

API functions

Progress DataDirect for ODBC drivers are Level 1 compliant, and support ODBC Core and Level 1 functions. As described in the following table, a limited set of Level 2 functions are also supported.

Table 6: Function Conformance for ODBC 2.x Applications

Core Functions	Level 1 Functions	Level 2 Functions
SQLAllocConnect[5]	SQLColumns	SQLBrowseConnect
SQLAllocEnv[5]	SQLDriverConnect	SQLDataSources
SQLAllocStmt[5]	SQLGetConnectOption	SQLDescribeParam[5]
SQLBindCol	SQLGetData	SQLExtendedFetch (forward scrolling only)
SQLBindParameter	SQLGetFunctions	SQLMoreResults
SQLCancel	SQLGetInfo	SQLNativeSql
SQLColAttributes	SQLGetStmtOption[5]	SQLNumParams
SQLConnect	SQLGetTypeInfo	SQLParamOptions[5]
SQLDescribeCol	SQLParamData	SQLSetScrollOptions
SQLDisconnect	SQLPutData	
SQLDrivers	SQLSetConnectOption	
SQLError	SQLSetStmtOption[5]	
SQLExecDirect	SQLSpecialColumns	
SQLExecute	SQLStatistics	
SQLFetch	SQLTables	
SQLFreeConnect[5]		
SQLFreeEnv[5]		
SQLFreeStmt		
SQLGetCursorName		
SQLNumResultCols		
SQLPrepare		
SQLRowCount		
SQLSetCursorName		
SQLTransact[5]		

The functions for ODBC 3.*x* Applications that the drivers support are listed in the following table. For any additions to these supported functions or differences in the support of specific functions, refer to "ODBC compliance" or "ODBC conformance" in the user's guide for your driver.

[5] For macOS, this function is not supported by the iODBC driver manager; therefore, it cannot currently be executed by the driver.

Table 7: Function Conformance for ODBC 3.x Applications

SQLAllocHandle	SQLGetDescField
SQLBindCol	SQLGetDescRec
SQLBindParameter	SQLGetDiagField
SQLBrowseConnect (except for Progress)	SQLGetDiagRec
SQLBulkOperations	SQLGetEnvAttr
SQLCancel	SQLGetFunctions
SQLCloseCursor	SQLGetInfo
SQLColAttribute	SQLGetStmtAttr
SQLColumns	SQLGetTypeInfo
SQLConnect	SQLMoreResults
SQLCopyDesc	SQLNativeSql
SQLDataSources	SQLNumParens
SQLDescribeCol	SQLNumResultCols
SQLDisconnect	SQLParamData
SQLDriverConnect	SQLPrepare
SQLDrivers	SQLPutData
SQLEndTran	SQLRowCount
SQLError	SQLSetConnectAttr
SQLExecDirect	SQLSetCursorName
SQLExecute	SQLSetDescField
SQLExtendedFetch	SQLSetDescRec
SQLFetch	SQLSetEnvAttr
SQLFetchScroll (forward scrolling only)	SQLSetStmtAttr
SQLFreeHandle	SQLSpecialColumns
SQLFreeStmt	SQLStatistics
SQLGetConnectAttr	SQLTables
SQLGetCursorName	SQLTransact
SQLGetData	

Scalar functions

This section lists the scalar functions that ODBC supports. Your database system may not support all these functions. Refer to the documentation for your database system to find out which functions are supported. Also, depending on the driver that you are using, all the scalar functions may not be supported. To check which scalar functions are supported by a driver, use the SQLGetInfo ODBC function.

You can use these scalar functions in SQL statements using the following syntax:

`{fn scalar-function}`

where *scalar-function* is one of the functions listed in the following tables. For example:

```
SELECT {fn UCASE(NAME)} FROM EMP
```

Table 8: Scalar Functions

String Functions	Numeric Functions	Timedate Functions	System Functions
ASCII	ABS	CURDATE	CURSESSIONID
BIT_LENGTH	ACOS	CURTIME	CURRENT_USER
CHAR	ASIN	DATEDIFF	DATABASE
CHAR_LENGTH	ATAN	DAYNAME	IDENTITY
CONCAT	ATAN2	DAYOFMONTH	USER
DIFFERENCE	BITAND	DAYOFWEEK	
HEXTORAW	BITOR	DAYOFYEAR	
INSERT	CEILING	EXTRACT	
LCASE	COS	HOUR	
LEFT	COT	MINUTE	
LENGTH	DEGREES	MONTH	
LOCATE	EXP	MONTHNAME	
LOWER	FLOOR	NOW	
LTRIM	LOG	QUARTER	
OCTET_LENGTH	LOG10	SECOND	
RAWTOHEX	MOD	WEEK	
REPEAT	PI	YEAR	
REPLACE	POWER	CURRENT_DATE	
RIGHT	RADIANS	CURRENT_TIME	
RTRIM	RAND	CURRENT_ TIMESTAMP	
SOUNDEX	ROUND		
SPACE	ROUNDMAGIC		

String Functions	Numeric Functions	Timedate Functions	System Functions
SUBSTR	SIGN		
SUBSTRING	SIN		
UCASE	SQRT		
UPPER	TAN		
	TRUNCATE		

String functions

The following table lists the string functions that ODBC supports.

The string functions listed accept the following arguments:

- *string_exp* can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type is SQL_CHAR, SQL_VARCHAR, or SQL_LONGVARCHAR.
- *start*, *length*, and *count* can be the result of another scalar function or a literal numeric value, where the underlying data type is SQL_TINYINT, SQL_SMALLINT, or SQL_INTEGER.

The string functions are one-based; that is, the first character in the string is character 1.

Character string literals must be surrounded in single quotation marks.

Table 9: Scalar String Functions

Function	Returns
ASCII(*string_exp*)	ASCII code value of the leftmost character of *string_exp* as an integer.
BIT_LENGTH(*string_exp*) *[ODBC 3.0 only]*	The length in bits of the string expression.
CHAR(*code*)	The character with the ASCII code value specified by *code*. *code* should be between 0 and 255; otherwise, the return value is data-source dependent.
CHAR_LENGTH(*string_exp*) *[ODBC 3.0 only]*	The length in characters of the string expression, if the string expression is of a character data type; otherwise, 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 CHARACTER_LENGTH function.)
CHARACTER_LENGTH(*string_exp*) *[ODBC 3.0 only]*	The length in characters of the string expression, if the string expression is of a character data type; otherwise, 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*)	The string resulting from concatenating *string_exp2* and *string_exp1*. The string is system dependent.

Function	Returns
DIFFERENCE(*string_exp1*, *string_exp2*)	An integer value that indicates the difference between the values returned by the SOUNDEX function for *string_exp1* and *string_exp2*.
INSERT(*string_exp1*, *start*, *length*, *string_exp2*)	A 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*)	Uppercase characters in *string_exp* converted to lowercase.
LEFT(*string_exp*,*count*)	The *count* of characters of *string_exp*.
LENGTH(*string_exp*)	The number of characters in *string_exp*, excluding trailing blanks and the string termination character.
LOCATE(*string_exp1*, *string_exp2*[,*start*])	The starting position of the first occurrence of *string_exp1* within *string_exp2*. If *start* is not specified, the search begins with the first character position in *string_exp2*. 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, 0 is returned.
LTRIM(*string_exp*)	The characters of *string_exp* with leading blanks removed.
OCTET_LENGTH(*string_exp*) *[ODBC 3.0 only]*	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 only]*	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*)	A string composed of *string_exp* repeated *count* times.
REPLACE(*string_exp1*, *string_exp2*, *string_exp3*)	Replaces all occurrences of *string_exp2* in *string_exp1* with *string_exp3*.
RIGHT(*string_exp*, *count*)	The rightmost *count* of characters in *string_exp*.
RTRIM(*string_exp*)	The characters of *string_exp* with trailing blanks removed.
SOUNDEX(*string_exp*)	A data source dependent string representing the sound of the words in *string_exp*.
SPACE(*count*)	A string consisting of *count* spaces.
SUBSTRING(*string_exp*, *start*, *length*)	A string derived from *string_exp* beginning at the character position *start* for *length* characters.
UCASE(*string_exp*)	Lowercase characters in *string_exp* converted to uppercase.

Numeric functions

The following table lists the numeric functions that ODBC supports.

The numeric functions listed accept the following arguments:

- *numeric_exp* can be a column name, a numeric literal, or the result of another scalar function, where the underlying data type is SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, or SQL_DOUBLE.
- *float_exp* can be a column name, a numeric literal, or the result of another scalar function, where the underlying data type is SQL_FLOAT.
- *integer_exp* can be a column name, a numeric literal, or the result of another scalar function, where the underlying data type is SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, or SQL_BIGINT.

Table 10: Scalar Numeric Functions

Function	Returns
ABS(*numeric_exp*)	Absolute value of *numeric_exp*.
ACOS(*float_exp*)	Arccosine of *float_exp* as an angle in radians.
ASIN(*float_exp*)	Arcsine of *float_exp* as an angle in radians.
ATAN(*float_exp*)	Arctangent of *float_exp* as an angle in radians.
ATAN2(*float_exp1*, *float_exp2*)	Arctangent of the x and y coordinates, specified by *float_exp1* and *float_exp2* as an angle in radians.
CEILING(*numeric_exp*)	Smallest integer greater than or equal to *numeric_exp*.
COS(*float_exp*)	Cosine of *float_exp* as an angle in radians.
COT(*float_exp*)	Cotangent of *float_exp* as an angle in radians.
DEGREES(*numeric_exp*)	Number if degrees converted from *numeric_exp* radians.
EXP(*float_exp*)	Exponential value of *float_exp*.
FLOOR(*numeric_exp*)	Largest integer less than or equal to *numeric_exp*.
LOG(*float_exp*)	Natural log of *float_exp*.
LOG10(*float_exp*)	Base 10 log of *float_exp*.
MOD(*integer_exp1*, *integer_exp2*)	Remainder of *integer_exp1* divided by *integer_exp2*.
PI()	Constant value of pi as a floating-point number.
POWER(*numeric_exp*, *integer_exp*)	Value of *numeric_exp* to the power of *integer_exp*.
RADIANS(*numeric_exp*)	Number of radians converted from *numeric_exp* degrees.

Function	Returns
RAND([*integer_exp*])	Random floating-point value using *integer_exp* as the optional seed value.
ROUND(*numeric_exp*, *integer_exp*)	*numeric_exp* rounded to *integer_exp* places right of the decimal (left of the decimal if *integer_exp* is negative).
SIGN(*numeric_exp*)	Indicator of the sign of *numeric_exp*. If *numeric_exp* < 0, -1 is returned. If *numeric_exp* = 0, 0 is returned. If *numeric_exp* > 0, 1 is returned.
SIN(*float_exp*)	Sine of *float_exp*, where *float_exp* is an angle in radians.
SQRT(*float_exp*)	Square root of *float_exp*.
TAN(*float_exp*)	Tangent of *float_exp*, where *float_exp* is an angle in radians.
TRUNCATE(*numeric_exp*, *integer_exp*)	*numeric_exp* truncated to *integer_exp* places right of the decimal. (If *integer_exp* is negative, truncation is to the left of the decimal.)

Date and time functions

The following table lists the date and time functions that ODBC supports.

The date and time functions listed accept the following arguments:

- *date_exp* can be a column name, a date or timestamp literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_DATE, or SQL_TIMESTAMP.
- *time_exp* can be a column name, a timestamp or timestamp literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_TIME, or SQL_TIMESTAMP.
- *timestamp_exp* can be a column name; a time, date, or timestamp literal; or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_TIME, SQL_DATE, or SQL_TIMESTAMP.

Table 11: Scalar Time and Date Functions

Function	Returns
CURRENT_DATE() *[ODBC 3.0 only]*	Current date.
CURRENT_TIME[(*time-precision*)] *[ODBC 3.0 only]*	Current local time. The *time-precision* argument determines the seconds precision of the returned value.
CURRENT_TIMESTAMP([*timestamp-precision*]) *[ODBC 3.0 only]*	Current local date and local time as a timestamp value. The *timestamp-precision* argument determines the seconds precision of the returned timestamp.

Function	Returns
CURDATE()	Current date as a date value.
CURTIME()	Current local time as a time value.
DAYNAME(*date_exp*)	Character string containing a data-source-specific name of the day for the day portion of *date_exp*.
DAYOFMONTH(*date_exp*)	Day of the month in *date_exp* as an integer value (1–31).
DAYOFWEEK(*date_exp*)	Day of the week in *date_exp* as an integer value (1–7).
DAYOFYEAR(*date_exp*)	Day of the year in *date_exp* as an integer value (1–366).
EXTRACT({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND} FROM *datetime_value*)	Any of the date and time terms can be extracted from *datetime_value*.
HOUR(*time_exp*)	Hour in *time_exp* as an integer value (0–23).
MINUTE(*time_exp*)	Minute in *time_exp* as an integer value (0–59).
MONTH(*date_exp*)	Month in *date_exp* as an integer value (1–12).
MONTHNAME(*date_exp*)	Character string containing the data source-specific name of the month.
NOW()	Current date and time as a timestamp value.
QUARTER(*date_exp*)	Quarter in *date_exp* as an integer value (1–4).
SECOND(*time_exp*)	Second in *date_exp* as an integer value (0–59).
TIMESTAMPADD(*interval*, *integer_exp*, *time_exp*)	Timestamp calculated by adding *integer_exp* intervals of type interval to *time_exp*. *interval* can be one of the following values: 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 Fractional seconds are expressed in billionths of a second.

Function	Returns
TIMESTAMPDIFF(*interval*, *time_exp1*, *time_exp2*)	Integer number of intervals of type *interval* by which *time_exp2* is greater than *time_exp1*. *interval* has the same values as TIMESTAMPADD. Fractional seconds are expressed in billionths of a second.
WEEK(*date_exp*)	Week of the year in *date_exp* as an integer value (1–53).
YEAR(*date_exp*)	Year in *date_exp*. The range is data-source dependent.

System functions

The following table lists the system functions that ODBC supports.

Table 12: Scalar System Functions

Function	Returns
DATABASE()	Name of the database, corresponding to the connection handle (*hdbc*).
IFNULL(*exp*,*value*)	*value*, if *exp* is null.
USER()	Authorization name of the user.

7

Internationalization, localization, and Unicode

This chapter provides an overview of how internationalization, localization, and Unicode relate to each other. It also provides a background on Unicode, and how it is accommodated by Unicode and non-Unicode ODBC drivers.

For details, see the following topics:

- Internationalization and Localization
- Unicode character encoding
- Unicode and non-Unicode ODBC drivers
- Driver Manager and Unicode encoding on UNIX/Linux
- Character encoding in the odbc.ini and odbcinst.ini files

Internationalization and Localization

Software that has been designed for *internationalization* is able to manage different linguistic and cultural conventions transparently and without modification. The same binary copy of an application should run on any localized version of an operating system without requiring source code changes.

Software that has been designed for *localization* includes language translation (such as text messages, icons, and buttons), cultural data (such as dates, times, and currency), and other components (such as input methods and spell checkers) for meeting regional market requirements.

Properly designed applications can accommodate a localized interface without extensive modification. The applications can be designed, first, to run internationally, and, second, to accommodate the language- and cultural-specific elements of a designated locale.

Locale

A locale represents the language and cultural data chosen by the user and dynamically loaded into memory at runtime. The locale settings are applied to the operating system and to subsequent application launches.

While language is a fairly straightforward item, cultural data is a little more complex. Dates, numbers, and currency are all examples of data that is formatted according to cultural expectations. Because cultural preferences are bound to a geographic area, country is an important element of locale. Together these two elements (language and country) provide a precise context in which information can be presented. Locale presents information in the language and form that is best understood and appreciated by the local user.

Language

A locale's language is specified by the ISO 639 standard. The following table lists some commonly used language codes.

Language Code	Language
en	English
nl	Dutch
fr	French
es	Spanish
zh	Chinese
ja	Japanese
vi	Vietnamese

Because language is correlated with geography, a language code might not capture all the nuances of usage in a particular area. For example, French and Canadian French may use different phrases and terms to mean different things even though basic grammar and vocabulary are the same. Language is only one element of locale.

Country

The locale's country identifier is also specified by an ISO standard, ISO 3166, which describes valid two-letter codes for all countries. ISO 3166 defines these codes in uppercase letters. The following table lists some commonly used country codes.

Country Code	Country
US	United States
FR	France
IE	Ireland
CA	Canada
MX	Mexico

The country code provides more contextual information for a locale and affects a language's usage, word spelling, and collation rules.

Variant

A variant is an optional extension to a locale. It identifies a custom locale that is not possible to create with just language and country codes. Variants can be used by anyone to add additional context for identifying a locale. The locale en_US represents English (United States), but en_US_CA represents even more information and might identify a locale for English (California, U.S.A). Operating system or software vendors can use these variants to create more descriptive locales for their specific environments.

Unicode character encoding

In addition to locale, the other major component of internationalizing software is the use of the Universal Codeset, or Unicode. Most developers know that Unicode is a standard encoding that can be used to support multilingual character sets. Unfortunately, understanding Unicode is not as simple as its name would indicate. Software developers have used a number of character encodings, from ASCII to Unicode, to solve the many problems that arise when developing software applications that can be used worldwide.

Background

Most legacy computing environments have used ASCII character encoding developed by the ANSI standards body to store and manipulate character strings inside software applications. ASCII encoding was convenient for programmers because each ASCII character could be stored as a byte. The initial version of ASCII used only 7 of the 8 bits available in a byte, which meant that applications could use only 128 different characters. This version of ASCII could not account for European characters and was completely inadequate for Asian characters. Using the eighth bit to extend the total range of characters to 256 added support for most European characters. Today, ASCII refers to either the 7-bit or 8-bit encoding of characters.

As the need increased for applications with additional international support, ANSI again increased the functionality of ASCII by developing an extension to accommodate multilingual software. The extension, known as the Double-Byte Character Set (DBCS), allowed existing applications to function without change, but provided for the use of additional characters, including complex Asian characters. With DBCS, characters map to either one byte (for example, American ASCII characters) or two bytes (for example, Asian characters). The DBCS environment also introduced the concept of an operating system code page that identified how characters would be encoded into byte sequences in a particular computing environment. DBCS encoding provided a cross-platform mechanism for building multilingual applications.

The DataDirect for ODBC UNIX, Linux, and macOS drivers can use double-byte character sets. The drivers normally use the character set defined by the default locale "C" unless explicitly pointed to another character set. The default locale "C" corresponds to the 7-bit US-ASCII character set. Use the following procedure to set the locale to a different character set:

1. Add the following line at the beginning of applications that use double-byte character sets:

   ```
   setlocale (LC_ALL, "");
   ```

   This is a standard function for UNIX-based platforms. It selects the character set indicated by the environment variable LANG as the one to be used by X/Open compliant, character-handling functions. If this line is not present, or if LANG is not set or is set to NULL, the default locale "C" is used.

2. Set the LANG environment variable to the appropriate character set. The command `locale -a` can be used to display all supported character sets on your system.

For more information, refer to the man pages for "locale" and "setlocale."

Using a DBCS, however, was not ideal; many developers felt that there was a better way to solve the problem. A group of leading software companies joined forces to form the Unicode Consortium. Together, they produced a new solution to building worldwide applications—Unicode. Unicode was originally designed as a fixed-width, uniform two-byte designation that could represent all modern scripts without the use of code pages. The Unicode Consortium has continued to evaluate new characters, and the current number of supported characters is over 109,000.

Although it seemed to be the perfect solution to building multilingual applications, Unicode started off with a significant drawback—it would have to be retrofitted into existing computing environments. To use the new paradigm, all applications would have to change. As a result, several standards-based transliterations were designed to convert two-byte fixed Unicode values into more appropriate character encodings, including, among others, UTF-8, UCS-2, UTF-16, and UTF-32.

UTF-8 is a standard method for transforming Unicode values into byte sequences that maintain transparency for all ASCII codes. UTF-8 is recognized by the Unicode Consortium as a mechanism for transforming Unicode values and is popular for use with HTML, XML, and other protocols. UTF-8 is, however, currently used primarily on AIX, HP-UX, Solaris, and Linux.

UCS-2 encoding is a fixed, two-byte encoding sequence and is a method for transforming Unicode values into byte sequences. It is the standard for Windows 95, Windows 98, Windows Me, and Windows NT.

UTF-16 is a superset of UCS-2, with the addition of some special characters in surrogate pairs. UTF-16 is the standard encoding for Windows 2000, Windows XP, Windows Vista, Windows Server 2003 and higher, and Windows 7 and higher. Microsoft recommends using UTF-16 for new applications.

UTF-32 encoding is a fixed-width, 4 byte method for transforming Unicode values into byte sequences. It is capable of defining all Unicode characters and is common for macOS platforms.

Refer to "Unicode Support" in the user's guide of your driver to determine which encoding formats your driver supports.

Unicode support in databases

Recently, database vendors have begun to support Unicode data types natively in their systems. With Unicode support, one database can hold multiple languages. For example, a large multinational corporation could store expense data in the local languages for the Japanese, U.S., English, German, and French offices in one database.

Not surprisingly, the implementation of Unicode data types varies from vendor to vendor. For example, the Microsoft SQL Server 2000 implementation of Unicode provides data in UTF-16 format, while Oracle provides Unicode data types in UTF-8 and UTF-16 formats. A consistent implementation of Unicode not only depends on the operating system, but also on the database itself.

Unicode support in ODBC

Prior to the ODBC 3.5 standard, all ODBC access to function calls and string data types was through ANSI encoding (either ASCII or DBCS). Applications and drivers were both ANSI-based.

The ODBC 3.5 standard specified that the ODBC Driver Manager be capable of mapping both Unicode function calls and string data types to ANSI encoding as transparently as possible. This meant that ODBC 3.5-compliant Unicode applications could use Unicode function calls and string data types with ANSI drivers because the Driver Manager could convert them to ANSI. Because of character limitations in ANSI, however, not all conversions are possible.

The ODBC Driver Manager version 3.5 and later, therefore, supports the following configurations:

- ANSI application with an ANSI driver

- ANSI application with a Unicode driver
- Unicode application with a Unicode driver
- Unicode application with an ANSI driver

A Unicode application can work with an ANSI driver because the Driver Manager provides limited Unicode-to-ANSI mapping. The Driver Manager makes it possible for a pre-3.5 ANSI driver to work with a Unicode application. What distinguishes a Unicode driver from a non-Unicode driver is the Unicode driver's capacity to interpret Unicode function calls without the intervention of the Driver Manager, as described in the following section.

Unicode and non-Unicode ODBC drivers

The way in which a driver handles function calls from a Unicode application determines whether it is considered a "Unicode driver."

Function calls

Instead of the standard ANSI SQL function calls, such as SQLConnect, Unicode applications use "W" (wide) function calls, such as SQLConnectW. If the driver is a true Unicode driver, it can understand "W" function calls and the Driver Manager can pass them through to the driver without conversion to ANSI. The DataDirect *for* ODBC drivers that support "W" function calls are:

- Aha!
- Amazon Redshift Wire Protocol
- Apache Hive
- Apache Cassandra
- Apache Spark SQL Wire Protocol
- Autonomous REST Connector
- GitHub
- Google Analytics
- Google BigQuery
- Greenplum Wire Protocol
- HubSpot
- IBM DB2 Wire Protocol
- Impala Wire Protocol
- Microsoft Dynamics 365
- Microsoft SharePoint
- MongoDB
- MySQL Wire Protocol
- Oracle
- Oracle Wire Protocol

- Oracle Service Cloud
- PostgreSQL Wire Protocol
- Progress OpenEdge Wire Protocol
- Salesforce
- SAP S/4HANA
- SQL Server Wire Protocol
- SQL Server Legacy Wire Protocol (UNIX only)
- Sybase Wire Protocol
- Sybase IQ Wire Protocol
- TeamCity
- Teradata
- XML

If a driver is a non-Unicode driver, it cannot understand W function calls, and the Driver Manager must convert them to ANSI calls before sending them to the driver. The Driver Manager determines the ANSI encoding system to which it must convert by referring to a code page. On Windows, this reference is to the Active Code Page. On non-Windows platforms, it is to the IANAAppCodePage connection string attribute, part of the `odbc.ini` file.

The following examples illustrate these conversion streams for the Progress DataDirect for ODBC drivers. The Driver Manager on UNIX and Linux determines the type of Unicode encoding of both the application and the driver, and performs conversions when the application and driver use different types of encoding. This determination is made by checking two ODBC attributes: SQL_ATTR_APP_UNICODE_TYPE and SQL_ATTR_DRIVER_UNICODE_TYPE, which can be set for either the environment, using SQLSetEnvAttr, or the connection, using SQLSetConnectAttr. "Driver Manager and Unicode Encoding on UNIX/Linux" describes in detail how this is done.

See also

Driver Manager and Unicode encoding on UNIX/Linux on page 70

Unicode application with a non-Unicode driver

An operation involving a Unicode application and a non-Unicode driver incurs more overhead because function conversion is involved.

Windows

1. The Unicode application sends UCS-2/UTF-16 function calls to the Driver Manager.
2. The Driver Manager converts the function calls from UCS-2/UTF-16 to ANSI. The type of ANSI is determined by the Driver Manager through reference to the client machine's Active Code Page.
3. The Driver Manager sends the ANSI function calls to the non-Unicode driver.
4. The driver returns ANSI argument values to the Driver Manager.
5. The Driver Manager converts the function calls from ANSI to UCS-2/UTF-16 and returns these converted calls to the application.

UNIX® and Linux

1. The Unicode application sends function calls to the Driver Manager. The Driver Manager expects the string arguments in these function calls to be UTF-8 or UTF-16 based on the value of the SQL_ATTR_APP_UNICODE_TYPE attribute. Note that the SQL_ATTR_APP_UNICODE_TYPE attribute can be set for the environment, using SQLSetEnvAttr, or the connection, using SQLSetConnectAttr.
2. The Driver Manager converts the function calls from UTF-8 or UTF-16 to ANSI. The type of ANSI is determined by the Driver Manager through reference to the client machine's value for the IANAAppCodePage connection string attribute.
3. The Driver Manager sends the converted ANSI function calls to the non-Unicode driver.
4. The driver returns ANSI argument values to the Driver Manager.
5. The Driver Manager converts the function calls from ANSI to UTF-8 or UTF-16 and returns these converted calls to the application.

macOS

On macOS, this scenario does not apply. All currently available drivers are Unicode drivers.

Unicode application with a Unicode driver

An operation involving a Unicode application and a Unicode driver that use the same Unicode encoding is efficient because no function conversion is involved. If the application and the driver each use different types of encoding, there is some conversion overhead. See "Driver Manager and Unicode Encoding on UNIX/Linux" for details.

Windows

1. The Unicode application sends UCS-2 or UTF-16 function calls to the Driver Manager.
2. The Driver Manager does not have to convert the UCS-2/UTF-16 function calls to ANSI. It passes the Unicode function call to the Unicode driver.
3. The driver returns UCS-2/UTF-16 argument values to the Driver Manager.
4. The Driver Manager returns UCS-2/UTF-16 function calls to the application.

UNIX and Linux

1. The Unicode application sends function calls to the Driver Manager. The Driver Manager expects the string arguments in these function calls to be UTF-8 or UTF-16 based on the value of the SQL_ATTR_APP_UNICODE_TYPE attribute. Note that the SQL_ATTR_APP_UNICODE_TYPE attribute can be set for the environment, using SQLSetEnvAttr, or the connection, using SQLSetConnectAttr.
2. The Driver Manager passes Unicode function calls to the Unicode driver. The Driver Manager has to perform function call conversions if the SQL_ATTR_APP_UNICODE_TYPE is different from the SQL_ATTR_DRIVER_UNICODE_TYPE.
3. The driver returns argument values to the Driver Manager. Whether these are UTF-8 or UTF-16 argument values is based on the value of the SQL_ATTR_DRIVER_UNICODE_TYPE attribute.
4. The Driver Manager returns appropriate function calls to the application based on the SQL_ATTR_APP_UNICODE_TYPE attribute value. The Driver Manager has to perform function call conversions if the SQL_ATTR_DRIVER_UNICODE_TYPE value is different from the SQL_ATTR_APP_UNICODE_TYPE value.

macOS

1. The Unicode application sends UTF-32 function calls to the Driver Manager.
2. The Driver Manager does not have to convert the UTF-32 function calls to ANSI. It passes the Unicode function call to the Unicode driver.
3. The driver returns UTF-32 argument values to the Driver Manager.
4. The Driver Manager returns UTF-32 function calls to the application.

See also

Driver Manager and Unicode encoding on UNIX/Linux on page 70

Data

ODBC C data types are used to indicate the type of C buffers that store data in the application. This is in contrast to SQL data types, which are mapped to native database types to store data in a database (data store). ANSI applications bind to the C data type SQL_C_CHAR and expect to receive information bound in the same way. Similarly, most Unicode applications bind to the C data type SQL_C_WCHAR (wide data type) and expect to receive information bound in the same way. Any ODBC 3.5-compliant Unicode driver must be capable of supporting SQL_C_CHAR and SQL_C_WCHAR so that it can return data to both ANSI and Unicode applications.

When the driver communicates with the database, it must use ODBC SQL data types, such as SQL_CHAR and SQL_WCHAR, that map to native database types. In the case of ANSI data and an ANSI database, the driver receives data bound to SQL_C_CHAR and passes it to the database as SQL_CHAR. The same is true of SQL_C_WCHAR and SQL_WCHAR in the case of Unicode data and a Unicode database.

When data from the application and the data stored in the database differ in format, for example, ANSI application data and Unicode database data, conversions must be performed. The driver cannot receive SQL_C_CHAR data and pass it to a Unicode database that expects to receive a SQL_WCHAR data type. The driver or the Driver Manager must be capable of converting SQL_C_CHAR to SQL_WCHAR, and vice versa.

The simplest cases of data communication are when the application, the driver, and the database are all of the same type and encoding, ANSI-to-ANSI-to-ANSI or Unicode-to-Unicode-to-Unicode. There is no data conversion involved in these instances.

When a difference exists between data types, a conversion from one type to another must take place at the driver or Driver Manager level, which involves additional overhead. The type of driver determines whether these conversions are performed by the driver or the Driver Manager. "Driver Manager and Unicode Encoding on UNIX/Linux" describes how the Driver Manager determines the type of Unicode encoding of the application and driver.

The following sections discuss two basic types of data conversion in the Progress DataDirect ODBC drivers and the Driver Manager. How an individual driver exchanges different types of data with a particular database at the database level is beyond the scope of this discussion.

See also

Driver Manager and Unicode encoding on UNIX/Linux on page 70

Unicode driver

The Unicode driver, not the Driver Manager, must convert SQL_C_CHAR (ANSI) data to SQL_WCHAR (Unicode) data, and vice versa, as well as SQL_C_WCHAR (Unicode) data to SQL_CHAR (ANSI) data, and vice versa.

The driver must use client code page information (Active Code Page on Windows and IANAAppCodePage attribute on UNIX/Linux/macOS) to determine which ANSI code page to use for the conversions. The Active Code Page or IANAAppCodePage must match the database default character encoding; if it does not, conversion errors are possible.

ANSI driver

The Driver Manager, not the ANSI driver, must convert SQL_C_WCHAR (Unicode) data to SQL_CHAR (ANSI) data, and vice versa (see "Unicode Support in ODBC" for a detailed discussion). This is necessary because ANSI drivers do not support any Unicode ODBC types.

The Driver Manager must use client code page information (Active Code Page on Windows and the IANAAppCodePage attribute on UNIX/Linux/macOS) to determine which ANSI code page to use for the conversions. The Active Code Page or IANAAppCodePage must match the database default character encoding. If not, conversion errors are possible.

See also

Unicode support in ODBC on page 64

Default Unicode mapping

The following table shows the default Unicode mapping for an application's SQL_C_WCHAR variables.

Platform	Default Unicode Mapping
Windows	UCS-2/UTF-16
AIX	UTF-8
HP-UX	UTF-8
Solaris	UTF-8
Linux	UTF-8
macOS	UTF-32

Connection attribute for Unicode

If you do not want to use the default Unicode mappings for SQL_C_WCHAR, a connection attribute is available to override the default mappings. This attribute determines how character data is converted and presented to an application and the database.

Attribute	Description
SQL_ATTR_APP_WCHAR_TYPE (1061)	Sets the SQL_C_WCHAR type for parameter and column binding to the Unicode type, either SQL_DD_CP_UTF16 (default for Windows) or SQL_DD_CP_UTF8 (default for UNIX/Linux).

You can set this attribute before or after you connect. After this attribute is set, all conversions are made based on the character set specified.

For example:

```
rc = SQLSetConnectAttr (hdbc, SQL_ATTR_APP_WCHAR_TYPE,
(void *)SQL_DD_CP_UTF16, SQL_IS_INTEGER);
```

SQLGetConnectAttr and SQLSetConnectAttr for the SQL_ATTR_APP_WCHAR_TYPE attribute return a SQL State of HYC00 for drivers that do not support Unicode.

This connection attribute and its valid values can be found in the file `qesqlext.h`, which is installed with the product.

Note: On Mac Platforms, the iODBC Driver Manager supports only UTF-32. As a result, this attribute is not currently supported.

Driver Manager and Unicode encoding on UNIX/Linux

UNIX®

Unicode ODBC drivers on UNIX and Linux can use UTF-8 or UTF-16 encoding. To use a single UTF-8 or UTF-16 application with either a UTF-8 or UTF-16 driver, the Driver Manager must be able to determine with which type of encoding the application and driver use and, if necessary, convert them accordingly.

To make this determination, the Driver Manager supports a set of ODBC attributes that can be set for the environment or the connection. If your application uses both UTF-8 and UTF-16 drivers in the same environment, encoding should be set for the connection only; otherwise, either method can be used.

- To configure the encoding type for the environment, set the ODBC environment attributes SQL_ATTR_APP_UNICODE_TYPE and SQL_ATTR_DRIVER_UNICODE_TYPE using SQLSetEnvAttr.
- To configure the encoding for the connection only, set the ODBC connection attribute SQL_ATTR_APP_UNICODE_TYPE and SQL_ATTR_DRIVER_UNICODE_TYPE using SQLSetConnectAttr.

The attributes support values of SQL_DD_CP_UTF8 and SQL_DD_CP_UTF16. The default value is SQL_DD_CP_UTF8.

Note: You must specify a value for SQL_ATTR_DRIVER_UNICODE_TYPE when using third-party drivers. However, for DataDirect drivers, the driver manager detects the Unicode type for the driver by default.

The Driver Manager performs the following steps before actually connecting to the driver.

1. Determine the application Unicode type: Applications that use UTF-16 encoding for their string types need to set SQL_ATTR_APP_UNICODE_TYPE accordingly at connection, or, if setting the encoding type for the environment, before connecting to any driver. When the Driver Manager reads this attribute, it expects all string arguments to the ODBC "W" functions to be in the specified Unicode format. This attribute also indicates how the SQL_C_WCHAR buffers must be encoded.
2. Determine the driver Unicode type: The Driver Manager must determine through which Unicode encoding the driver supports its "W" functions. This is done as follows:
   a. SQLGetEnvAttr(SQL_ATTR_DRIVER_UNICODE_TYPE) or SQLGetConnectATTR (SQL_ATTR_DRIVER_UNICODE_TYPE) is called in the driver by the Driver Manager. The driver, if capable, returns either SQL_DD_CP_UTF16 or SQL_DD_CP_UTF8 to indicate to the Driver Manager which encoding it expects.
   b. If the preceding call to SQLGetEnvAttr fails, the Driver Manager looks either in the Data Source section of the `odbc.ini` specified by the connection string or in the connection string itself for a connection

option named DriverUnicodeType. Valid values for this option are 1 (UTF-16) or 2 (UTF-8). The Driver Manager assumes that the Unicode encoding of the driver corresponds to the value specified.

c. If neither of the preceding attempts are successful, the Driver Manager assumes that the Unicode encoding of the driver is UTF-8.

3. Determine if the driver supports SQL_ATTR_WCHAR_TYPE: SQLSetConnectAttr (SQL_ATTR_WCHAR_TYPE, *x*) is called in the driver by the Driver Manager, where *x* is either SQL_DD_CP_UTF8 or SQL_DD_CP_UTF16, depending on the value of the SQL_ATTR_APP_UNICODE_TYPE setting. If the driver returns any error on this call to SQLSetConnectAttr, the Driver Manager assumes that the driver does not support this connection attribute.

If an error occurs, the Driver Manager returns a warning. The Driver Manager does not convert all bound parameter data from the application Unicode type to the driver Unicode type specified by SQL_ATTR_DRIVER_UNICODE_TYPE. Neither does it convert all data bound as SQL_C_WCHAR to the application Unicode type specified by SQL_ATTR_APP_UNICODE_TYPE.

Based on the information it has gathered prior to connection, the Driver Manager either does not have to convert function calls, or, before calling the driver, it converts to either UTF-8 or UTF-16 all string arguments to calls to the ODBC "W" functions.

References

The Java Tutorials, http://docs.oracle.com/javase/tutorial/i18n/index.html

Unicode Support in the Solaris Operating Environment, May 2000, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303-4900

Character encoding in the odbc.ini and odbcinst.ini files

UNIX®

The `odbc.ini` and `odbcinst.ini` files can use ANSI or UTF-8 encoding. To ensure encoding compatibility between these files and the application, the Driver Manager converts encoding when necessary. This allows applications with different encoding to write to or read from the `odbc.ini` or `odbcinst.ini` file using the following functions:

ANSI functions:

- SQLWritePrivateProfileString
- SQLGetPrivateProfileString

Unicode (wide or "W") functions:

- SQLWritePrivateProfileStringW
- SQLGetPrivateProfileStringW

For the Driver Manager to accomplish this task, it must determine the encoding format your application and file use. How the Driver Manager makes this determination is dependent on the encoding of the function called by the application.

When a Unicode function is called, the Driver Manager assumes that the `odbc.ini` and `odbcinst.ini` files use UTF-8 encoding, while encoding for the application is determined by the ODBC_App_Unicode_Type variable in the system environment:

- If the variable is set to `ODBC_App_Unicode_Type=1`, the Driver Manager expects that application uses input and output strings encoded as UTF-16. When the application calls SQLWritePrivateProfileStringW, the Driver Manager converts UTF-16 input strings and writes them as UTF-8 in the file. When the application calls SQLGetPrivateProfileStringW, the Driver Manager returns the requested values using UTF-16 encoding.
- If any other value is specified for ODBC_App_Unicode_Type, or if the variable is not defined, the Driver Manager assumes that the application and file use UTF-8. When this occurs, the Driver Manager does not convert strings passed between the application and file.

When an ANSI function is called, the Driver Manager assumes that application and file use ANSI encoding. In this scenario, the Driver Manger does not convert strings passed between the application and file.

For more information about the `odbc.ini` and `odbcinst.ini` files, refer to "Configuring the Product on UNIX/Linux" in the user's guide for you driver.

8

Designing ODBC applications for performance optimization

Developing performance-oriented ODBC applications is not easy. Microsoft's *ODBC Programmer's Reference* does not provide information about system performance. In addition, ODBC drivers and the ODBC driver manager do not return warnings when applications run inefficiently. This chapter contains some general guidelines that have been compiled by examining the ODBC implementations of numerous shipping ODBC applications. These guidelines include:

- Use catalog functions appropriately
- Retrieve only required data
- Select functions that optimize performance
- Manage connections and updates

Following these general rules will help you solve some common ODBC performance problems, such as those listed in the following table.

Table 13: Common Performance Problems Using ODBC Applications

Problem	Solution	See guidelines in...
Network communication is slow.	Reduce network traffic.	"Using Catalog Functions"
The process of evaluating complex SQL queries on the database server is slow and can reduce concurrency.	Simplify queries.	"Using Catalog Functions" "Selecting ODBC Functions"

Problem	Solution	See guidelines in...
Excessive calls from the application to the driver slow performance.	Optimize application-to-driver interaction.	"Retrieving Data" "Selecting ODBC Functions"
Disk I/O is slow.	Limit disk input/output.	"Managing Connections and Updates"

For details, see the following topics:

- Using catalog functions
- Retrieving data
- Selecting ODBC functions
- Managing connections and updates

Using catalog functions

Because catalog functions, such as those listed here, are slow compared to other ODBC functions, their frequent use can impair system performance:

- SQLColumns
- SQLForeignKeys
- SQLGetTypeInfo
- SQLSpecialColumns
- SQLStatistics
- SQLTables

SQLGetTypeInfo is included in this list of expensive ODBC functions because many drivers must query the server to obtain accurate information about which types are supported (for example, to find dynamic types such as user defined types).

Caching information to minimize the use of catalog functions

To return all result column information mandated by the ODBC specification, a driver may have to perform multiple queries, joins, subqueries, or unions to return the required result set for a single call to a catalog function. These particular elements of the SQL language are performance expensive.

Although it is almost impossible to write an ODBC application without catalog functions, their use should be minimized. By caching information, applications can avoid multiple executions.

For example, call SQLGetTypeInfo once in the application and cache the elements of the result set that your application depends on. It is unlikely that any application uses all elements of the result set generated by a catalog function, so the cached information should not be difficult to maintain.

Avoiding search patterns

Passing NULL arguments or search patterns to catalog functions generates time-consuming queries. In addition, network traffic potentially increases because of unwanted results. Always supply as many non-NULL arguments to catalog functions as possible. Because catalog functions are slow, applications should invoke them efficiently. Any information that the application can send the driver when calling catalog functions can result in improved performance and reliability.

For example, consider a call to SQLTables where the application requests information about the table "Customers." Often, this call is coded as shown, using as many NULL arguments as possible:

```
rc = SQLTables (hstmt, NULL, 0, NULL, 0, "Customers", SQL_NTS, NULL, 0);
```

A driver processes this SQLTables call into SQL that looks like this:

```
SELECT ...  FROM SysTables WHERE TableName = 'Customers'
UNION ALL
SELECT ... FROM SysViews WHERE ViewName = 'Customers'
UNION ALL
SELECT ... FROM SysSynonyms WHERE SynName = 'Customers' ORDER BY ...
```

In our example, the application provides scant information about the object for which information was requested. Suppose three "Customers" tables were returned in the result set: the first table owned by the user named Beth, the second owned by the sales department, and the third a view created by management.

It may not be obvious to the end user which table to choose. If the application had specified the OwnerName argument in the SQLTables call, only one table would be returned and performance would improve. Less network traffic would be required to return only one result row and unwanted rows would be filtered by the database. In addition, if the TableType argument was supplied, the SQL sent to the server can be optimized from a three-query union into a single Select statement as shown:

```
SELECT ... FROM SysTables WHERE TableName = 'Customers' AND Owner = 'Beth'
```

Using a dummy query to determine table characteristics

Avoid using SQLColumns to determine characteristics about a table. Instead, use a dummy query with SQLDescribeCol.

Consider an application that allows the user to choose the columns that will be selected. Should the application use SQLColumns to return information about the columns to the user or prepare a dummy query and call SQLDescribeCol?

Case 1: SQLColumns Method

```
rc = SQLColumns (... "UnknownTable" ...);
// This call to SQLColumns will generate a query to the system catalogs...
// possibly a join which must be prepared, executed, and produce a result set
rc = SQLBindCol (...);
rc = SQLExtendedFetch (...);
// user must retrieve N rows from the server
// N = # result columns of UnknownTable
// result column information has now been obtained
```

Case 2: SQLDescribeCol Method

```
// prepare dummy query
rc = SQLPrepare (... "SELECT * FROM UnknownTable WHERE 1 = 0" ...);
// query is never executed on the server - only prepared
rc = SQLNumResultCols (...);
for (irow = 1; irow <= NumColumns; irow++) {
```

```
   rc = SQLDescribeCol (...)
   // + optional calls to SQLColAttributes
   }
// result column information has now been obtained
// Note we also know the column ordering within the table!
// This information cannot be assumed from the SQLColumns example.
```

In both cases, a query is sent to the server, but in Case 1, the query must be evaluated and form a result set that must be sent to the client. Clearly, Case 2 is the better performing model.

To complicate this discussion, let us consider a database server that does not natively support preparing a SQL statement. The performance of Case 1 does not change, but the performance of Case 2 improves slightly because the dummy query is evaluated before being prepared. Because the Where clause of the query always evaluates to FALSE, the query generates no result rows and should execute without accessing table data. Again, for this situation, Case 2 outperforms Case 1.

Retrieving data

To retrieve data efficiently, return only the data that you need, and choose the most efficient method of doing so. The guidelines in this section will help you optimize system performance when retrieving data with ODBC applications.

Retrieving long data

Because retrieving long data across the network is slow and resource-intensive, applications should not request long data (SQL_LONGVARCHAR, SQL_WLONGVARCHAR, and SQL_LONGVARBINARY data) unless it is necessary.

Most users do not want to see long data. If the user does need to see these result items, the application can query the database again, specifying only long columns in the select list. This technique allows the average user to retrieve the result set without having to pay a high performance penalty for network traffic.

Although the best approach is to exclude long data from the select list, some applications do not formulate the select list before sending the query to the ODBC driver (that is, some applications simply `SELECT * FROM` *`table_name`* `...`). If the select list contains long data, the driver must retrieve that data at fetch time even if the application does not bind the long data in the result set. When possible, use a technique that does not retrieve all columns of the table.

Reducing the size of data retrieved

Sometimes, long data must be retrieved. When this is the case, remember that most users do not want to see 100 KB, or more, of text on the screen.

To reduce network traffic and improve performance, you can reduce the size of data being retrieved to some manageable limit by calling SQLSetStmtAttr with the SQL_ATTR_MAX_LENGTH option.

Eliminating SQL_LONGVARCHAR, SQL_WLONGVARCHAR, and SQL_LONGVARBINARY data from the result set is ideal for optimizing performance.

Many application developers mistakenly assume that if they call SQLGetData with a container of size *x* that the ODBC driver only retrieves *x* bytes of information from the server. Because SQLGetData can be called multiple times for any one column, most drivers optimize their network use by retrieving long data in large chunks and then returning it to the user when requested. For example:

```
char CaseContainer[1000];
...
rc = SQLExecDirect (hstmt, "SELECT CaseHistory FROM Cases WHERE CaseNo = 71164", SQL_NTS);
...
rc = SQLFetch (hstmt);
rc = SQLGetData (hstmt, 1, CaseContainer,(SWORD) sizeof(CaseContainer), ...);
```

At this point, it is more likely that an ODBC driver will retrieve 64 KB of information from the server instead of 1 KB. In terms of network access, one 64-KB retrieval is less expensive than 64 retrievals of 1 KB. Unfortunately, the application may not call SQLGetData again; therefore, the first and only retrieval of CaseHistory would be slowed by the fact that 64 KB of data must be sent across the network.

Many ODBC drivers allow you to limit the amount of data retrieved across the network by supporting the SQL_MAX_LENGTH attribute. This attribute allows the driver to communicate to the database server that only *x* bytes of data are relevant to the client. The server responds by sending only the first *x* bytes of data for all result columns. This optimization substantially reduces network traffic and improves client performance. The previous example returned only one row, but consider the case where 100 rows are returned in the result set—the performance improvement would be substantial.

Using bound columns

Retrieving data through bound columns (SQLBindCol) instead of using SQLGetData reduces the ODBC call load and improves performance.

Consider the following code fragment:

```
rc = SQLExecDirect (hstmt, "SELECT <20 columns> FROM Employees WHERE HireDate >= ?", SQL_NTS);
do {
   rc  = SQLFetch (hstmt);
   // call SQLGetData 20 times
} while ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO));
```

Suppose the query returns 90 result rows. In this case, 1891 ODBC calls are made (20 calls to SQLGetData x 90 result rows + 91 calls to SQLFetch).

Consider the same scenario that uses SQLBindCol instead of SQLGetData:

```
rc = SQLExecDirect (hstmt, "SELECT <20 columns> FROM Employees WHERE HireDate >= ?", SQL_NTS);
// call SQLBindCol 20 times
do {
rc  = SQLFetch (hstmt);
} while ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO));
```

The number of ODBC calls made is reduced from 1891 to 111 (20 calls to SQLBindCol + 91 calls to SQLFetch). In addition to reducing the call load, many drivers optimize how SQLBindCol is used by binding result information directly from the database server into the user's buffer. That is, instead of the driver retrieving information into a container and then copying that information to the user's buffer, the driver simply requests the information from the server be placed directly into the user's buffer.

Using SQLExtendedFetch instead of SQLFetch

Use SQLExtendedFetch to retrieve data instead of SQLFetch. The ODBC call load decreases (resulting in better performance) and the code is less complex (resulting in more maintainable code).

Most ODBC drivers now support SQLExtendedFetch for forward only cursors; yet, most ODBC applications use SQLFetch to retrieve data. Consider the examples in "Using Bound Columns", this time using SQLExtendedFetch instead of SQLFetch:

```
rc = SQLSetStmtOption (hstmt, SQL_ROWSET_SIZE, 100);
// use arrays of 100 elements
rc = SQLExecDirect (hstmt, "SELECT <20 columns> FROM Employees WHERE HireDate >= ?", SQL_NTS);
// call SQLBindCol 1 time specifying row-wise binding
do {
```

```
   rc = SQLExtendedFetch (hstmt, SQL_FETCH_NEXT, 0, &RowsFetched,RowStatus);
} while ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO));
```

Notice the improvement from the previous examples. The initial call load was 1891 ODBC calls. By choosing ODBC calls carefully, the number of ODBC calls made by the application has now been reduced to 4 (1 SQLSetStmtOption + 1 SQLExecDirect + 1 SQLBindCol + 1 SQLExtendedFetch). In addition to reducing the call load, many ODBC drivers retrieve data from the server in arrays, further improving the performance by reducing network traffic.

For ODBC drivers that do not support SQLExtendedFetch, the application can enable forward-only cursors using the ODBC cursor library:

```
 (rc=SQLSetConnectOption (hdbc, SQL_ODBC_CURSORS, SQL_CUR_USE_IF_NEEDED);
```

Although using the cursor library does not improve performance, it should not be detrimental to application response time when using forward-only cursors (no logging is required). Furthermore, using the cursor library means that the application can always depend on SQLExtendedFetch being available. This simplifies the code because the application does not require two algorithms (one using SQLExtendedFetch and one using SQLFetch).

See also

Using bound columns on page 77

Choosing the right data type

Retrieving and sending certain data types can be expensive. When you are working with data on a large scale, select the data type that can be processed most efficiently. For example, integer data is processed faster than floating-point data. Floating-point data is defined according to internal database-specific formats, usually in a compressed format. The data must be decompressed and converted into a different format so that it can be processed by the wire protocol.

Selecting ODBC functions

The guidelines in this section will help you select which ODBC functions will give you the best performance.

Using SQLPrepare/SQLExecute and SQLExecDirect

Using SQLPrepare/SQLExecute is not always as efficient as SQLExecDirect. Use SQLExecDirect for queries that will be executed once and SQLPrepare/SQLExecute for queries that will be executed multiple times.

ODBC drivers are optimized based on the perceived use of the functions that are being executed. SQLPrepare/SQLExecute is optimized for multiple executions of statements that use parameter markers. SQLExecDirect is optimized for a single execution of a SQL statement. Unfortunately, more than 75% of all ODBC applications use SQLPrepare/SQLExecute exclusively.

Consider the case where an ODBC driver implements SQLPrepare by creating a stored procedure on the server that contains the prepared statement. Creating stored procedures involve substantial overhead, but the statement can be executed multiple times. Although creating stored procedures is performance-expensive, execution is minimal because the query is parsed and optimization paths are stored at create procedure time.

Using SQLPrepare/SQLExecute for a statement that is executed only once results in unnecessary overhead. Furthermore, applications that use SQLPrepare/SQLExecute for large single execution query batches exhibit poor performance. Similarly, applications that always use SQLExecDirect do not perform as well as those that use a logical combination of SQLPrepare/SQLExecute and SQLExecDirect sequences.

Using arrays of parameters

Passing arrays of parameter values for bulk insert operations, for example, with SQLPrepare/SQLExecute and SQLExecDirect can reduce the ODBC call load and network traffic. To use arrays of parameters, the application calls SQLSetStmtAttr with the following attribute arguments:

- SQL_ATTR_PARAMSET_SIZE sets the array size of the parameter.
- SQL_ATTR_PARAMS_PROCESSED_PTR assigns a variable filled by SQLExecute, which contains the number of rows that are actually inserted.
- SQL_ATTR_PARAM_STATUS_PTR points to an array in which status information for each row of parameter values is returned.

Note: ODBC 3.*x* replaced the ODBC 2.*x* call to SQLParamOptions with calls to SQLSetStmtAttr using the SQL_ATTR_PARAMSET_SIZE, SQL_ATTR_PARAMS_PROCESSED_ARRAY, and SQL_ATTR_PARAM_STATUS_PTR arguments.

Before executing the statement, the application sets the value of each data element in the bound array. When the statement is executed, the driver tries to process the entire array contents using one network roundtrip. For example, let us compare the following examples, Case 1 and Case 2.

Case 1: Executing Prepared Statement Multiple Times

```
rc = SQLPrepare (hstmt, "INSERT INTO DailyLedger (...) VALUES (?,?,...)", SQL_NTS);
// bind parameters
...
do {
   // read ledger values into bound parameter buffers
   ...
   rc = SQLExecute (hstmt);
   // insert row
} while ! (eof);
```

Case 2: Using Arrays of Parameters

```
SQLPrepare (hstmt, " INSERT INTO DailyLedger (...) VALUES (?,?,...)", SQL_NTS);
SQLSetStmtAttr (hstmt, SQL_ATTR_PARAMSET_SIZE, (UDWORD)100, SQL_IS_UINTEGER);
SQLSetStmtAttr (hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, &rows_processed, SQL_IS_POINTER);
// Specify an array in which to return the status of
// each set of parameters.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_STATUS_PTR, ParamStatusArray, SQL_IS_POINTER);
// pass 100 parameters per execute
// bind parameters
...
do {
   // read up to 100 ledger values into
   // bound parameter buffers
   ...
   rc = SQLExecute (hstmt);
   // insert a group of 100 rows
} while ! (eof);
```

In Case 1, if there are 100 rows to insert, 101 network roundtrips are required to the server, one to prepare the statement with SQLPrepare and 100 additional roundtrips for each time SQLExecute is called.

In Case 2, the call load has been reduced from 100 SQLExecute calls to only 1 SQLExecute call. Furthermore, network traffic is reduced considerably.

Using the cursor library

If the driver provides scrollable cursors, do not use the cursor library. The cursor library creates local temporary log files, which are performance-expensive to generate and provide worse performance than native scrollable cursors.

The cursor library adds support for static cursors, which simplifies the coding of applications that use scrollable cursors. However, the cursor library creates temporary log files on the user's local disk drive to accomplish the task. Typically, disk I/O is a slow operation. Although the cursor library is beneficial, applications should not automatically choose to use the cursor library when an ODBC driver supports scrollable cursors natively.

Typically, ODBC drivers that support scrollable cursors achieve high performance by requesting that the database server produce a scrollable result set instead of emulating the capability by creating log files. Many applications use:

```
rc = SQLSetConnectOption (hdbc, SQL_ODBC_CURSORS, SQL_CUR_USE_ODBC);
```

but should use:

```
rc = SQLSetConnectOption (hdbc, SQL_ODBC_CURSORS, SQL_CUR_USE_IF_NEEDED);
```

Managing connections and updates

The guidelines in this section will help you to manage connections and updates to improve system performance for your ODBC applications.

Managing connections

Connection management is important to application performance. Optimize your application by connecting once and using multiple statement handles, instead of performing multiple connections. Avoid connecting to a data source after establishing an initial connection.

Although gathering driver information at connect time is a good practice, it is often more efficient to gather it in one step rather than two steps. Some ODBC applications are designed to call informational gathering routines that have no record of already attached connection handles. For example, some applications establish a connection and then call a routine in a separate DLL or shared library that reattaches and gathers information about the driver. Applications that are designed as separate entities should pass the already connected HDBC pointer to the data collection routine instead of establishing a second connection.

Another bad practice is to connect and disconnect several times throughout your application to process SQL statements. Connection handles can have multiple statement handles associated with them. Statement handles can provide memory storage for information about SQL statements. Therefore, applications do not need to allocate new connection handles to process SQL statements. Instead, applications should use statement handles to manage multiple SQL statements.

You can significantly improve performance with connection pooling, especially for applications that connect over a network or through the World Wide Web. With connection pooling, closing connections does not close the physical connection to the database. When an application requests a connection, an active connection from the connection pool is reused, avoiding the network round trips needed to create a new connection.

Connection and statement handling should be addressed before implementation. Spending time and thoughtfully handling connection management improves application performance and maintainability.

Managing commits in transactions

Committing data is extremely disk I/O intensive and slow. If the driver can support transactions, always turn autocommit off.

What does a commit actually involve? The database server must flush back to disk every data page that contains updated or new data. This is not a sequential write but a searched write to replace existing data in the table. By default, autocommit is on when connecting to a data source. Autocommit mode usually impairs system performance because of the significant amount of disk I/O needed to commit every operation.

Some database servers do not provide an Autocommit mode. For this type of server, the ODBC driver must explicitly issue a COMMIT statement and a BEGIN TRANSACTION for every operation sent to the server. In addition to the large amount of disk I/O required to support Autocommit mode, a performance penalty is paid for up to three network requests for every statement issued by an application.

Although using transactions can help application performance, do not take this tip too far. Leaving transactions active can reduce throughput by holding locks on rows for long times, preventing other users from accessing the rows. Commit transactions in intervals that allow maximum concurrency.

Choosing the right transaction model

Many systems support distributed transactions; that is, transactions that span multiple connections. Distributed transactions are at least four times slower than normal transactions due to the logging and network round trips necessary to communicate between all the components involved in the distributed transaction. Unless distributed transactions are required, avoid using them. Instead, use local transactions when possible.

Using positioned updates and deletes

Use positioned updates and deletes or SQLSetPos to update data. Although positioned updates do not apply to all types of applications, developers should use positioned updates and deletes when it makes sense. Positioned updates (either through `UPDATE WHERE CURRENT OF CURSOR` or through SQLSetPos) allow the developer to signal the driver to "change the data here" by positioning the database cursor at the appropriate row to be changed. The designer is not forced to build a complex SQL statement, but simply supplies the data to be changed.

In addition to making the application more maintainable, positioned updates usually result in improved performance. Because the database server is already positioned on the row for the Select statement in process, performance-expensive operations to locate the row to be changed are not needed. If the row must be located, the server typically has an internal pointer to the row available (for example, ROWID).

Using SQLSpecialColumns

Use SQLSpecialColumns to determine the optimal set of columns to use in the Where clause for updating data. Often, pseudo-columns provide the fastest access to the data, and these columns can only be determined by using SQLSpecialColumns.

Some applications cannot be designed to take advantage of positioned updates and deletes. These applications typically update data by forming a Where clause consisting of some subset of the column values returned in the result set. Some applications may formulate the Where clause by using all searchable result columns or by calling SQLStatistics to find columns that are part of a unique index. These methods typically work, but can result in fairly complex queries.

Consider the following example:

```
rc = SQLExecDirect (hstmt, "SELECT first_name, last_name, ssn, address, city, state, zip
 FROM emp", SQL_NTS);
// fetchdata
...
rc = SQLExecDirect (hstmt, "UPDATE EMP SET ADDRESS = ? WHERE first_name = ? AND last_name
 = ? AND
   ssn = ? AND address = ? AND city = ? AND state = ? AND zip = ?", SQL_NTS);
// fairly complex query
```

Applications should call SQLSpecialColumns/SQL_BEST_ROWID to retrieve the optimal set of columns (possibly a pseudo-column) that identifies a given record. Many databases support special columns that are not explicitly defined by the user in the table definition but are "hidden" columns of every table (for example, ROWID and TID). These pseudo-columns provide the fastest access to data because they typically point to the exact location of the record. Because pseudo-columns are not part of the explicit table definition, they are not returned from SQLColumns. To determine if pseudo-columns exist, call SQLSpecialColumns.

Consider the previous example again:

```
...
rc = SQLSpecialColumns (hstmt, ..... 'emp', ...);
...
rc = SQLExecDirect (hstmt, "SELECT first_name, last_name, ssn, address, city, state,
zip, ROWID
   FROM emp", SQL_NTS);
// fetch data and probably "hide" ROWID from the user
...
rc = SQLExecDirect (hstmt, "UPDATE emp SET address = ? WHERE ROWID = ?",SQL_NTS);
// fastest access to the data!
```

If your data source does not contain special pseudo-columns, the result set of SQLSpecialColumns consists of columns of the optimal unique index on the specified table (if a unique index exists). Therefore, your application does not need to call SQLStatistics to find the smallest unique index.

9

Using indexes

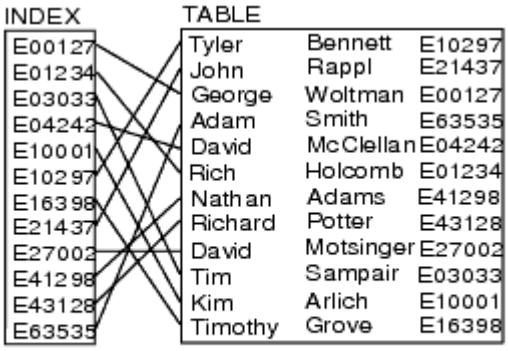
This chapter discusses the ways in which you can improve the performance of database activity using indexes. It provides general guidelines that apply to most databases. Consult your database vendor's documentation for more detailed information.

For details, see the following topics:

Introduction

An index is a database structure that you can use to improve the performance of database activity. A database table can have one or more indexes associated with it.

An index is defined by a field expression that you specify when you create the index. Typically, the field expression is a single field name, like emp_id. An index created on the emp_id field, for example, contains a sorted list of the employee ID values in the table. Each value in the list is accompanied by references to the rows that contain that value.



A database driver can use indexes to find rows quickly. An index on the emp_id field, for example, greatly reduces the time that the driver spends searching for a particular employee ID value. Consider the following Where clause:

```
WHERE EMP_id = 'E10001'
```

Without an index, the server must search the entire database table to find those rows having an employee ID of E10001. By using an index on the emp_id field, however, the server can quickly find those rows.

Indexes may improve the performance of SQL statements. You may not notice this improvement with small tables, but it can be significant for large tables; however, there can be disadvantages to having too many indexes. Indexes can slow down the performance of some inserts, updates, and deletes when the driver has to maintain the indexes as well as the database tables. Also, indexes take additional disk space.

Improving row selection performance

For indexes to improve the performance of selections, the index expression must match the selection condition exactly. For example, if you have created an index whose expression is last_name, the following Select statement uses the index:

```
SELECT * FROM emp WHERE last_name = 'Smith'
```

This Select statement, however, does not use the index:

```
SELECT * FROM emp WHERE UPPER(last_name) = 'SMITH'
```

The second statement does not use the index because the Where clause contains UPPER(last_name), which does not match the index expression last_name. If you plan to use the UPPER function in all your Select statements and your database supports indexes on expressions, then you should define an index using the expression UPPER(last_name).

Indexing multiple fields

If you often use Where clauses that involve more than one field, you may want to build an index containing multiple fields. Consider the following Where clause:

```
WHERE last_name = 'Smith' AND first_name = 'Thomas'
```

For this condition, the optimal index field expression is last_name, first_name. This creates a concatenated index.

Concatenated indexes can also be used for Where clauses that contain only the first of two concatenated fields. The last_name, first_name index also improves the performance of the following Where clause (even though no first name value is specified):

```
last_name = 'Smith'
```

Consider the following Where clause:

```
WHERE last_name = 'Smith' AND middle_name = 'Edward' AND first_name = 'Thomas'
```

If your index fields include all the conditions of the Where clause in that order, the driver can use the entire index. If, however, your index is on two nonconsecutive fields, for example, last_name and first_name, the driver can use only the last_name field of the index.

The driver uses only one index when processing Where clauses. If you have complex Where clauses that involve a number of conditions for different fields and have indexes on more than one field, the driver chooses an index to use. The driver attempts to use indexes on conditions that use the equal sign as the relational operator rather than conditions using other operators (such as greater than). Assume you have an index on the emp_id field as well as the last_name field and the following Where clause:

```
WHERE emp_id >= 'E10001' AND last_name = 'Smith'
```

In this case, the driver selects the index on the last_name field.

If no conditions have the equal sign, the driver first attempts to use an index on a condition that has a lower *and* upper bound, and then attempts to use an index on a condition that has a lower *or* upper bound. The driver always attempts to use the most restrictive index that satisfies the Where clause.

In most cases, the driver does not use an index if the Where clause contains an OR comparison operator. For example, the driver does not use an index for the following Where clause:

```
WHERE emp_id >= 'E10001' OR last_name = 'Smith'
```

Deciding which indexes to create

Before you create indexes for a database table, consider how you will use the table. The most common operations on a table are:

- Inserting, updating, and deleting rows
- Retrieving rows

If you most often insert, update, and delete rows, then the fewer indexes associated with the table, the better the performance. This is because the driver must maintain the indexes as well as the database tables, thus slowing down the performance of row inserts, updates, and deletes. It may be more efficient to drop all indexes before modifying a large number of rows, and re-create the indexes after the modifications.

If you most often retrieve rows, you must look further to define the criteria for retrieving rows and create indexes to improve the performance of these retrievals. Assume you have an employee database table and you will retrieve rows based on employee name, department, or hire date. You would create three indexes—one on the dept field, one on the hire_date field, and one on the last_name field. Or perhaps, for the retrievals based on the name field, you would want an index that concatenates the last_name and the first_name fields (see "Indexing Multiple Fields" for details).

Here are a few rules to help you decide which indexes to create:

- If your row retrievals are based on only one field at a time (for example, `dept='D101'`), create an index on these fields.
- If your row retrievals are based on a combination of fields, look at the combinations.

- If the comparison operator for the conditions is And (for example, `city = 'Raleigh' AND state = 'NC'`), then build a concatenated index on the city and state fields. This index is also useful for retrieving rows based on the city field.
- If the comparison operator is OR (for example, `dept = 'D101' OR hire_date > {01/30/89}`), an index does not help performance. Therefore, you need not create one.
- If the retrieval conditions contain both AND and OR comparison operators, you can use an index if the OR conditions are grouped. For example:

  ```
  dept = 'D101' AND (hire_date > {01/30/89} OR exempt = 1)
  ```

  In this case, an index on the dept field improves performance.
- If the AND conditions are grouped, an index does not improve performance. For example:

  ```
  (dept = 'D101' AND hire_date) > {01/30/89}) OR exempt = 1
  ```

See also

Indexing multiple fields on page 84

Improving join performance

When joining database tables, index tables can greatly improve performance. Unless the proper indexes are available, queries that use joins can take a long time.

Assume you have the following Select statement:

```
SELECT * FROM dept, emp WHERE dept.dept_id = emp.dept_id
```

In this example, the `dept` and `emp` database tables are being joined using the `dept_id` field. When the driver executes a query that contains a join, it processes the tables from left to right and uses an index on the second table's join field (the `dept` field of the `emp` table). To improve join performance, you need an index on the join field of the second table in the FROM clause.

If the FROM clause includes a third table, the driver also uses an index on the field in the third table that joins it to any previous table. For example:

```
SELECT * FROM dept, emp, addr WHERE dept.dept_id = emp.dept AND emp.loc = addr.loc
```

In this case, you should have an index on the emp.dept field and the addr.loc field.

10

Locking and isolation levels

This chapter discusses locking and isolation levels and how their settings can affect the data you retrieve.

For details, see the following topics:

- Locking
- Isolation levels
- Locking modes and levels

Locking

Locking is a database operation that restricts a user from accessing a table or record. Locking is used in situations where more than one user might try to use the same table or record at the same time. By locking the table or record, the system ensures that only one user at a time can affect the data.

Locking is critical in multiuser databases, where different users can try to access or modify the same records concurrently. Although such concurrent database activity is desirable, it can create problems. Without locking, for example, if two users try to modify the same record at the same time, they might encounter problems ranging from retrieving bad data to deleting data that the other user needs. If, however, the first user to access a record can lock that record to temporarily prevent other users from modifying it, such problems can be avoided. Locking provides a way to manage concurrent database access while minimizing the various problems it can cause.

Isolation levels

An isolation level represents a particular locking strategy employed in the database system to improve data consistency. The higher the isolation level, the more complex the locking strategy behind it. The isolation level provided by the database determines whether a transaction will encounter the following behaviors in data consistency:

Dirty reads	User 1 modifies a row. User 2 reads the same row before User 1 commits. User 1 performs a rollback. User 2 has read a row that has never really existed in the database. User 2 may base decisions on false data.
Non-repeatable reads	User 1 reads a row, but does not commit. User 2 modifies or deletes the same row and then commits. User 1 rereads the row and finds it has changed (or has been deleted).
Phantom reads	User 1 uses a search condition to read a set of rows, but does not commit. User 2 inserts one or more rows that satisfy this search condition, then commits. User 1 rereads the rows using the search condition and discovers rows that were not present before.

Isolation levels represent the database system's ability to prevent these behaviors. The American National Standards Institute (ANSI) defines four isolation levels:

- Read uncommitted (0)
- Read committed (1)
- Repeatable read (2)
- Serializable (3)

In ascending order (0–3), these isolation levels provide an increasing amount of data consistency to the transaction. At the lowest level, all three behaviors can occur. At the highest level, none can occur. The success of each level in preventing these behaviors is due to the locking strategies they use, which are as follows:

Read uncommitted (0)	Locks are obtained on modifications to the database and held until end of transaction (EOT). Reading from the database does not involve any locking.
Read committed (1)	Locks are acquired for reading and modifying the database. Locks are released after reading but locks on modified objects are held until EOT.
Repeatable read (2)	Locks are obtained for reading and modifying the database. Locks on all modified objects are held until EOT. Locks obtained for reading data are held until EOT. Locks on non-modified access structures (such as indexes and hashing structures) are released after reading.
Serializable (3)	All data read or modified is locked until EOT. All access structures that are modified are locked until EOT. Access structures used by the query are locked until EOT.

The following table shows what data consistency behaviors can occur at each isolation level.

Table 14: Isolation Levels and Data Consistency

Level	Dirty Read	Nonrepeatable Read	Phantom Read
0, Read uncommitted	Yes	Yes	Yes

Level	Dirty Read	Nonrepeatable Read	Phantom Read
1, Read committed	No	Yes	Yes
2, Repeatable read	No	No	Yes
3, Serializable	No	No	No

Although higher isolation levels provide better data consistency, this consistency can be costly in terms of the concurrency provided to individual users. Concurrency is the ability of multiple users to access and modify data simultaneously. As isolation levels increase, so does the chance that the locking strategy used will create problems in concurrency.

The higher the isolation level, the more locking involved, and the more time users may spend waiting for data to be freed by another user. Because of this inverse relationship between isolation levels and concurrency, you must consider how people use the database before choosing an isolation level. You must weigh the trade-offs between data consistency and concurrency, and decide which is more important.

Locking modes and levels

Different database systems use various locking modes, but they have two basic modes in common: shared and exclusive. Shared locks can be held on a single object by multiple users. If one user has a shared lock on a record, then a second user can also get a shared lock on that same record; however, the second user cannot get an exclusive lock on that record. Exclusive locks are exclusive to the user that obtains them. If one user has an exclusive lock on a record, then a second user cannot get either type of lock on the same record.

Performance and concurrency can also be affected by the locking level used in the database system. The locking level determines the size of an object that is locked in a database. For example, many database systems let you lock an entire table, as well as individual records. An intermediate level of locking, page-level locking, is also common. A page contains one or more records and is typically the amount of data read from the disk in a single disk access. The major disadvantage of page-level locking is that if one user locks a record, a second user may not be able to lock other records because they are stored on the same page as the locked record.

11

SSL encryption cipher suites

The following tables list the SSL/TLS encryption cipher suites supported by Progress DataDirect ODBC drivers. The driver attempts to negotiate either SSL v3 or TLS v1 with the server using OpenSSL cipher suites.

Note: For information about using SSL/TLS data encryption with the drivers, refer to "Using security" in the user's guide for your driver.

OpenSSL Cipher Suites to SSL v2 Cipher Suites

The following table shows the OpenSSL encryption cipher suites that a driver can use if it can negotiate SSL v2 with the server, with the name of the corresponding SSL v2 encryption cipher suites.

OpenSSL Cipher Suite	SSL Encryption Cipher Suite
DES-CBC-MD5	SSL_CK_DES_64_CBC_WITH_MD5
DES-CBC3-MD5	SSL_CK_DES_192_EDE3_CBC_WITH_MD5
EXP-RC2-CBC-MD5	SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5
EXP-RC4-MD5	SSL_CK_RC4_128_EXPORT40_WITH_MD5
RC2-CBC-MD5	SSL_CK_RC2_128_CBC_WITH_MD5
RC4-MD5	SSL_CK_RC4_128_WITH_MD5

Mapping OpenSSL Cipher Suites to SSL v3 Cipher Suites

The following table shows the OpenSSL encryption cipher suites that a driver can use if it can negotiate SSL v3 with the server, with the name of the corresponding SSL v3 encryption cipher suites.

OpenSSL Cipher Suite	SSL v3 Cipher Suite
AES128-GCM-SHA256	TLS_RSA_WITH_AES_128_GCM_SHA256
AES128-SHA	TLS_RSA_WITH_AES_128_CBC_SHA[6]
AES128-SHA256	TLS_RSA_WITH_AES_128_CBC_SHA256
AES256-GCM-SHA384	TLS_RSA_WITH_AES_256_GCM_SHA384
AES256-SHA	TLS_RSA_WITH_AES_256_CBC_SHA [6]
AES256-SHA256	TLS_RSA_WITH_AES_256_CBC_SHA256
DES-CBC3-SHA	SSL_RSA_WITH_3DES_EDE_CBC_SHA
DES-CBC-SHA	SSL_RSA_WITH_DES_CBC_SHA
DHE-DSS-AES128-GCM-SHA256	TLS_DHE_DSS_WITH_AES_128_GCM_SHA256
DHE-DSS-AES128-SHA	TLS_DHE_DSS_WITH_AES_128_CBC_SHA[6]
DHE-DSS-AES128-SHA256	TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
DHE-DSS-AES256-GCM-SHA384	TLS_DHE_DSS_WITH_AES_256_GCM_SHA384
DHE-DSS-AES256-SHA	TLS_DHE_DSS_WITH_AES_256_CBC_SHA[6]
DHE-DSS-AES256-SHA256	TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
DHE-DSS-SEED-SHA	TLS_DHE_DSS_WITH_SEED_CBC_SHA[7]
DHE-RSA-AES128-GCM-SHA256	TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
DHE-RSA-AES128-SHA	TLS_DHE_RSA_WITH_AES_128_CBC_SHA[6]
DHE-RSA-AES128-SHA256	TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
DHE-RSA-AES256-GCM-SHA384	TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
DHE-RSA-AES256-SHA	TLS_DHE_RSA_WITH_AES_256_CBC_SHA[6]
DHE-RSA-AES256-SHA256	TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
DHE-RSA-SEED-SHA	TLS_DHE_RSA_WITH_SEED_CBC_SHA[7]
EDH-DSS-DES-CBC3-SHA	SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA

[6] AES cipher suites from RFC3268 are used to extend TLS v1.

OpenSSL Cipher Suite	SSL v3 Cipher Suite
EDH-DSS-DES-CBC-SHA	SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
EDH-RSA-DES-CBC3-SHA	SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
EDH-RSA-DES-CBC-SHA	SSL_DHE_RSA_WITH_DES_CBC_SHA
EXP-DES-CBC-SHA	SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
EXP-EDH-DSS-DES-CBC-SHA	SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
EXP-EDH-RSA-DES-CBC-SHA	SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
EXP-RC2-CBC-MD5	SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
EXP-RC4-MD5	SSL_RSA_EXPORT_WITH_RC4_40_MD5
PSK-3DES-EDE-CBC-SHA	TLS_PSK_WITH_3DES_EDE_CBC_SHA
PSK-AES128-CBC-SHA	TLS_PSK_WITH_AES_128_CBC_SHA
PSK-AES256-CBC-SHA	TLS_PSK_WITH_AES_256_CBC_SHA
PSK-RC4-SHA	TLS_PSK_WITH_RC4_128_SHA
RC4-MD5	SSL_RSA_WITH_RC4_128_MD5
RC4-SHA	SSL_RSA_WITH_RC4_128_SHA
SEED-SHA	TLS_RSA_WITH_SEED_CBC_SHA[7]
SRP-3DES-EDE-CBC-SHA	TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA
SRP-AES-128-CBC-SHA	TLS_SRP_SHA_WITH_AES_128_CBC_SHA
SRP-AES-256-CBC-SHA	TLS_SRP_SHA_WITH_AES_256_CBC_SHA
SRP-DSS-3DES-EDE-CBC-SHA	TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA
SRP-DSS-AES-128-CBC-SHA	TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA
SRP-DSS-AES-256-CBC-SHA	TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA
SRP-RSA-3DES-EDE-CBC-SHA	TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA
SRP-RSA-AES-128-CBC-SHA	TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA
SRP-RSA-AES-256-CBC-SHA	TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA

[7] Seed cipher suites from RFC4162 are used to extend TLS v1.

Mapping OpenSSL Encryption Cipher Suites to TLS v1.0, TLS v1.1, and TLS v1.2 Cipher Suites

The following table shows the OpenSSL Encryption Cipher suites that a driver can use if it can negotiate TLS v1.0, TLS v1.1, and TLS v1.2 with the server, with the name of the corresponding cipher suites.

OpenSSL Cipher Suite	Maps to TLS v1 Cipher Suite
AES128-GCM-SHA256	TLS_RSA_WITH_AES_128_GCM_SHA256
AES128-SHA	TLS_RSA_WITH_AES_128_CBC_SHA[6]
AES128-SHA256	TLS_RSA_WITH_AES_128_CBC_SHA256
AES256-GCM-SHA384	TLS_RSA_WITH_AES_256_GCM_SHA384
AES256-SHA	TLS_RSA_WITH_AES_256_CBC_SHA[6]
AES256-SHA256	TLS_RSA_WITH_AES_256_CBC_SHA256
DES-CBC3-SHA	TLS_RSA_WITH_3DES_EDE_CBC_SHA
DES-CBC-SHA	TLS_RSA_WITH_DES_CBC_SHA
DHE-DSS-AES128-GCM-SHA256	DHE-DSS-AES128-GCM-SHA256
DHE-DSS-AES128-SHA	TLS_DHE_DSS_WITH_AES_128_CBC_SHA[6]
DHE-DSS-AES128-SHA256	TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
DHE-DSS-AES256-GCM-SHA384	TLS_DHE_DSS_WITH_AES_256_GCM_SHA384
DHE-DSS-AES256-SHA	TLS_DHE_DSS_WITH_AES_256_CBC_SHA[6]
DHE-DSS-AES256-SHA256	TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
DHE-DSS-SEED-SHA	TLS_DHE_DSS_WITH_SEED_CBC_SHA[7]
DHE-RSA-AES128-GCM-SHA256	TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
DHE-RSA-AES128-SHA	TLS_DHE_RSA_WITH_AES_128_CBC_SHA[6]
DHE-RSA-AES128-SHA	TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
DHE-RSA-AES256-GCM-SHA384	TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
DHE-RSA-AES256-SHA	TLS_DHE_RSA_WITH_AES_256_CBC_SHA[6]
DHE-RSA-AES256-SHA256	TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
DHE-RSA-SEED-SHA	TLS_DHE_RSA_WITH_SEED_CBC_SHA[7]
ECDHE-RSA-AES256-SHA384	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384

OpenSSL Cipher Suite	Maps to TLS v1 Cipher Suite
ECDHE-RSA-AES256-SHA	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
ECDHE-RSA-AES128-SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
ECDHE-RSA-AES128-SHA	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
EDH-DSS-DES-CBC3-SHA	TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
EDH-DSS-DES-CBC-SHA	TLS_DHE_DSS_WITH_DES_CBC_SHA
EDH-RSA-DES-CBC3-SHA	TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
EDH-RSA-DES-CBC-SHA	TLS_DHE_RSA_WITH_DES_CBC_SHA
EXP-DES-CBC-SHA	TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
EXP-EDH-DSS-DES-CBC-SHA	TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
EXP-EDH-RSA-DES-CBC-SHA	TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
EXP-RC2-CBC-MD5	TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
EXP-RC4-MD5	TLS_RSA_EXPORT_WITH_RC4_40_MD5
PSK-3DES-EDE-CBC-SHA	TLS_PSK_WITH_3DES_EDE_CBC_SHA
PSK-AES128-CBC-SHA	TLS_PSK_WITH_AES_128_CBC_SHA
PSK-AES256-CBC-SHA	TLS_PSK_WITH_AES_256_CBC_SHA
PSK-RC4-SHA	TLS_PSK_WITH_RC4_128_SHA
RC4-MD5	TLS_RSA_WITH_RC4_128_MD5
RC4-SHA	TLS_RSA_WITH_RC4_128_SHA
SEED-SHA	TLS_RSA_WITH_SEED_CBC_SHA[7]
SRP-3DES-EDE-CBC-SHA	TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA
SRP-AES-128-CBC-SHA	TLS_SRP_SHA_WITH_AES_128_CBC_SHA
SRP-AES-128-CBC-SHA	TLS_SRP_SHA_WITH_AES_128_CBC_SHA
SRP-AES-256-CBC-SHA	TLS_SRP_SHA_WITH_AES_256_CBC_SHA
SRP-DSS-3DES-EDE-CBC-SHA	TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA
SRP-DSS-AES-128-CBC-SHA	TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA
SRP-DSS-AES-256-CBC-SHA	TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA

OpenSSL Cipher Suite	Maps to TLS v1 Cipher Suite
SRP-RSA-3DES-EDE-CBC-SHA	TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA
SRP-RSA-AES-128-CBC-SHA	TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA
SRP-RSA-AES-256-CBC-SHA	TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA

Reference

OpenSSL Cryptography and SSL/TLS Toolkit

12

DataDirect Bulk Load

This chapter contains detailed information about the functions and statement attributes associated with DataDirect Bulk Load.

For a discussion of the operation of DataDirect Bulk Load for a specific driver, refer to "Using DataDirect Bulk Load" in the user's guide for the driver.

For details, see the following topics:

- DataDirect Bulk Load functions
- Utility functions
- Export, validate, and load functions
- DataDirect Bulk Load statement attributes

DataDirect Bulk Load functions

The following DataDirect functions and parameters are not part of the standard ODBC API. They include functions for returning errors and warnings on bulk operations as well as functions for bulk export, loading, and verification:

- SetBulkOperation (Salesforce driver only) on page 110
- GetBulkOperation (Salesforce driver only) on page 111

Note: For your application to use DataDirect Bulk Load functionality, it must obtain driver connection handles and function pointers, as follows:

---

1. Use SQLGetInfo with the parameter SQL_DRIVER_HDBC to obtain the driver's connection handle from the Driver Manager.
2. Use SQLGetInfo with the parameter SQL_DRIVER_HLIB to obtain the driver's shared library or DLL handle from the Driver Manager.
3. Obtain function pointers to the bulk load functions using the function name resolution method specific to your operating system. The bulk.c source file shipped with the drivers contains the function resolveName that illustrates how to obtain function pointers to the bulk load functions.

All of this is detailed in the code examples shown in the following sections. All of these functions can be found in the commented bulk.c source file that ships with the drivers. This file is located in the \samples\bulk subdirectory of the product installation directory along with a text file named bulk.txt. Please consult bulk.txt for instructions about the bulk.c file.

Utility functions

The example code in this section shows utility functions to which the DataDirect functions for bulk exporting, verification, and bulk loading refer, as well as the DataDirect functions GetBulkDiagRec and GetBulkDiagRecW.

GetBulkDiagRec and GetBulkDiagRecW

Syntax

```
SQLReturn
GetBulkDiagRec     (SQLSMALLINT  HandleType,
                    SQLHANDLE     Handle,
                    SQLSMALLINT   RecNumber,
                    SQLCHAR*      Sqlstate,
                    SQLINTEGER*   NativeError,
                    SQLCHAR*      MessageText,
                    SQLSMALLINT   BufferLength,
                    SQLSMALLINT*  TextLength);
GetBulkDiagRecW    (SQLSMALLINT  HandleType,
                    SQLHANDLE     Handle,
                    SQLSMALLINT   RecNumber,
                    SQLWCHAR*     Sqlstate,
                    SQLINTEGER*   NativeError,
                    SQLWCHAR*     MessageText,
                    SQLSMALLINT   BufferLength,
                    SQLSMALLINT*  TextLength);
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, SQL_NO_DATA, and SQL_ERROR.

Description

GetBulkDiagRec (ANSI application) and GetBulkDiagRecW (Unicode application) return errors and warnings generated by bulk operations. The argument definition, return values, and function behavior is the same as for the standard ODBC SQLGetDiagRec and SQLGetDiagRecW functions with the following exceptions:

- The GetBulkDiagRec and GetBulkDiagRecW functions can be called after a bulk load, export or validate function is invoked to retrieve any error messages generated by the bulk operation. Calling these functions after any function except a bulk function is not recommended.
- The values returned in the Sqlstate and MessageText buffers by the GetBulkDiagRecW function are encoded as UTF-16 on Windows platforms. On UNIX and Linux platforms, the values returned for Sqlstate and MessageText are UTF-16 if the value of the SQL_ATTR_APP_UNICODE_TYPE is SQL_DD_CP_UTF16 and UTF-8 if the value of SQL_ATTR_APP_UNICODE_TYPE is SQL_DD_CP_UTF8.
- The handle passed as the Handle argument must be a driver connection handle obtained by calling SQLGetInfo (<ODBC Conn Handle>, SQL_DRIVER_HDBC).
- SQL_HANDLE_DBC is the only value accepted for HandleType. Any other value causes an error to be returned.

Example

```
#include "qesqlext.h"

#ifndef NULL
#define NULL 0
#endif

#if (! defined (_WIN32)) && (! defined (_WIN64))
typedef void * HMODULE;
#endif

/* Get the address of a routine in a shared library or DLL. */
void * resolveName (
  HMODULE hmod,
  const char *name)
{
#if defined (_WIN32) || defined (_WIN64)

  return GetProcAddress (hmod, name);
#elif defined (hpux)
  void *routine = shl_findsym (hmod, name);

  shl_findsym (hmod, name, TYPE_PROCEDURE, &routine);

  return routine;
#else
  return dlsym (hmod, name);
#endif
}
/* Get errors directly from the driver's connection handle. */
void driverError (void *driverHandle, HMODULE hmod)
{
  UCHAR sqlstate[16];
  UCHAR errmsg[SQL_MAX_MESSAGE_LENGTH * 2];
  SDWORD nativeerr;
  SWORD actualmsglen;
  RETCODE rc;
  SQLSMALLINT  i;
  PGetBulkDiagRec getBulkDiagRec;

  getBulkDiagRec = (PGetBulkDiagRec)
    resolveName (hmod, "GetBulkDiagRec");

  if (! getBulkDiagRec) {
    printf ("Cannot find GetBulkDiagRec!\n");
```

```
    return;
  }


  i = 1;
loop:   rc = (*getBulkDiagRec) (SQL_HANDLE_DBC,
          driverHandle, i++,
          sqlstate, &nativeerr, errmsg,
          SQL_MAX_MESSAGE_LENGTH - 1, &actualmsglen);

  if (rc == SQL_ERROR) {
    printf ("GetBulkDiagRec failed!\n");
    return;
  }

  if (rc == SQL_NO_DATA_FOUND) return;

  printf ("SQLSTATE = %s\n", sqlstate);
  printf ("NATIVE ERROR = %d\n", nativeerr);
  errmsg[actualmsglen] = '\0';
  printf ("MSG = %s\n\n", errmsg);
  goto loop;
}
```

Export, validate, and load functions

The example code in this section shows the DataDirect functions for bulk exporting, verification, and bulk loading.

ExportTableToFile and ExportTableToFileW

Syntax

```
SQLReturn
ExportTableToFile  (HDBC      hdbc,
                    SQLCHAR*  TableName,
                    SQLCHAR*  FileName,
                    SQLLEN    IANAAppCodePage,
                    SQLLEN    ErrorTolerance,
                    SQLLEN    WarningTolerance,
                    SQLCHAR*  LogFile)
ExportTableToFileW (HDBC      hdbc,
                    SQLWCHAR* TableName,
                    SQLWCHAR* FileName,
                    SQLLEN    IANAAppCodePage,
                    SQLLEN    ErrorTolerance,
                    SQLLEN    WarningTolerance,
                    SQLWCHAR* LogFile)
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, and SQL_ERROR.

Purpose

ExportTableToFile (ANSI application) and ExportTableToFileW (Unicode application) bulk export a table to a physical file. Both a bulk data file and a bulk configuration file are produced by this operation. The configuration file has the same name as the data file, but with an XML extension. The bulk export operation can create a log file and can also export to external files. Refer to "External overflow files" in the user's guide for your driver for more information. The export operation can be configured such that if any errors or warnings occur:

- The operation always completes
- The operation always terminates
- The operation terminates after a certain threshold of warnings or errors is exceeded.

Parameters

hdbc

is the driver's connection handle, which is not the handle returned by SQLAllocHandle or SQLAllocConnect. To obtain the driver's connection handle, the application must then use the standard ODBC function SQLGetInfo (ODBC Conn Handle, SQL_DRIVER_HDBC).

TableName

is a null-terminated string that specifies the name of the source database table that contains the data to be exported.

FileName

is a null-terminated string that specifies the path (relative or absolute) and file name of the bulk load data file to which the data is to be exported. It also specifies the file name of the bulk configuration file. The file name must be the fully qualified path to the bulk data file. This file must not already exist. If the file already exists, an error is returned.

IANAAppCodePage

specifies the code page value to which the driver must convert all data for storage in the bulk data file. See "Code page values" for details about IANAAppCodePage. Refer to "Character Set Conversions" in the user's guide for your driver for more information.

The default value on Windows is the current code page of the machine. On UNIX, Linux, and macOS the default value is 4.

ErrorTolerance

specifies the number of errors to tolerate before an operation terminates. A value of 0 indicates that no errors are tolerated; the operation fails when the first error is encountered. The default of -1 means that an infinite number of errors is tolerated. WarningTolerance specifies the number of warnings to tolerate before an operation terminates. A value of 0 indicates that no warnings are tolerated; the operation fails when the first warning is encountered.

The default of -1 means that an infinite number of warnings is tolerated.

LogFile

is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk log file. Events logged to this file are:

- Total number of rows fetched

- A message for each row that failed to export
- Total number of rows that failed to export
- Total number of rows successfully exported

Information about the load is written to this file, preceded by a header. Information about the next load is appended to the end of the file.

If LogFile is NULL, no log file is created.

Example

```
HDBC      hdbc;
HENV      henv;
void      *driverHandle;
HMODULE      hmod;
PExportTableToFile exportTableToFile;

char      tableName[128];
char      fileName[512];
char      logFile[512];
int       errorTolerance;
int       warningTolerance;
int       codePage;

/* Get the driver's connection handle from the DM. This handle must be used when calling
 directly into the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HDBC, &driverHandle, 0, NULL);
if (rc != SQL_SUCCESS) {
    ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
    EnvClose (henv, hdbc);
    exit (255);
}

/* Get the DM's shared library or DLL handle to the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HLIB, &hmod, 0, NULL);
if (rc != SQL_SUCCESS) {
    ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
    EnvClose (henv, hdbc);
    exit (255);
}

exportTableToFile = (PExportTableToFile)
    resolveName (hmod, "ExportTableToFile");
if (! exportTableToFile) {
    printf ("Cannot find ExportTableToFile!\n");
    exit (255);
}

rc = (*exportTableToFile) (
      driverHandle,
      (const SQLCHAR *) tableName,
      (const SQLCHAR *) fileName,
      codePage,
      errorTolerance, warningTolerance,
      (const SQLCHAR *) logFile);
if (rc == SQL_SUCCESS) {
      printf ("Export succeeded.\n");
}else {
      driverError (driverHandle, hmod);
}
```

See also

Code page values on page 43

ValidateTableFromFile and ValidateTableFromFileW

Syntax

```
SQLReturn
ValidateTableFromFile  (HDBC       hdbc,
                        SQLCHAR*   TableName,
                        SQLCHAR*   ConfigFile,
                        SQLCHAR*   MessageList,
                        SQLULEN    MessageListSize,
                        SQLULEN*   NumMessages)
ValidateTableFromFileW (HDBC       hdbc,
                        SQLCHAR*   TableName,
                        SQLCHAR*   ConfigFile,
                        SQLCHAR*   MessageList,
                        SQLULEN    MessageListSize,
                        SQLULEN*   NumMessages)
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, and SQL_ERROR.

Purpose

ValidateTableFromFile (ANSI application) and ValidateTablefromFileW (Unicode application) verify the metadata in the configuration file against the data structure of the target database table. Refer to "Verification of the bulk load configuration file" in the user's guide for your driver for more detailed information.

Note: The Salesforce driver does not support ValidateTableFromFile and ValidateTableFromFileW.

Parameters

hdbc

is the driver's connection handle, which is not the handle returned by SQLAllocHandle or SQLAllocConnect. To obtain the driver's connection handle, the application must then use the standard ODBC function SQLGetInfo (*ODBC Conn Handle, SQL_DRIVER_HDBC*).

TableName

is a null-terminated character string that specifies the name of the target database table into which the data is to be loaded.

ConfigFile

is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk configuration file.

MessageList

specifies a pointer to a buffer used to record any of the errors and warnings. MessageList must not be null.

MessageListSize

specifies the maximum number of characters that can be written to the buffer to which MessageList points. If the buffer to which MessageList points is not big enough to hold all of the messages generated by the validation process, the validation is aborted and SQL_ERROR is returned.

NumMessages

contains the number of messages that were added to the buffer. This method reports the following criteria:

- Check data types - Each column data type is checked to ensure no loss of data occurs. If a data type mismatch is detected, the driver adds an entry to the MessageList in the following format: `Risk of data conversion loss: Destination column_number is of type x, and source column_number is of type y`.
- Check column sizes - Each column is checked for appropriate size. If column sizes are too small in destination tables, the driver adds an entry to the MessageList in the following format: `Possible Data Truncation: Destination column_number is of size x while source column_number is of size y`.
- Check codepages - Each column is checked for appropriate code page alignment between the source and destination. If a mismatch occurs, the driver adds an entry to the MessageList in the following format: `Destination column code page for column_number risks data corruption if transposed without correct character conversion from source column_number`.
- Check Config Col Info - The destination metadata and the column metadata in the configuration file are checked for consistency of items such as length for character and binary data types, the character encoding code page for character types, precision and scale for numeric types, and nullablity for all types. If any inconsistency is found, the driver adds an entry to the MessageList in the following format: `Destination column metadata for column_number has column info mismatches from source column_number`.
- Check Column Names and Mapping - The columns defined in the configuration file are compared to the destination table columns based on the order of the columns. If the number of columns in the configuration file and/or import file does not match the number of columns in the table, the driver adds an entry to the MessageList in the following format: `The number of destination columns number does not match the number of source columns number`.

The function returns an array of null-terminated strings in the buffer to which MessageList points with an entry for each of these checks. If the driver determines that the information in the bulk load configuration file matches the metadata of the destination table, a return code of SQL_SUCCESS is returned and the MessageList remains empty.

If the driver determines that there are minor differences in the information in the bulk load configuration file and the destination table, then SQL_SUCCESS_WITH_INFO is returned and the MessageList is populated with information on the cause of the potential problems.

If the driver determines that the information in the bulk load information file cannot successfully be loaded into the destination table, then a return code of SQL_ERROR is returned and the MessageList is populated with information on the problems and mismatches between the source and destination.

Example

```
HDBC      hdbc;
HENV      henv;
void      *driverHandle;
HMODULE   hmod;
PValidateTableFromFile validateTableFromFile;

char      tableName[128];
char      configFile[512];
char      messageList[10240];
SQLLEN    numMessages;

/* Get the driver's connection handle from the DM. This handle must be used when calling
 directly into the driver. */
```

```
rc = SQLGetInfo (hdbc, SQL_DRIVER_HDBC, &driverHandle, 0, NULL);
if (rc != SQL_SUCCESS) {
    ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
    EnvClose (henv, hdbc);
    exit (255);
}

/* Get the DM's shared library or DLL handle to the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HLIB, &hmod, 0, NULL);
if (rc != SQL_SUCCESS) {
    ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
    EnvClose (henv, hdbc);
    exit (255);
}

validateTableFromFile = (PValidateTableFromFile)
  resolveName (hmod, "ValidateTableFromFile");
if (!validateTableFromFile) {
  printf ("Cannot find ValidateTableFromFile!\n");
  exit (255);
}

messageList[0] = 0;
numMessages = 0;

rc = (*validateTableFromFile) (
      driverHandle,
      (const SQLCHAR *) tableName,
      (const SQLCHAR *) configFile,
      (SQLCHAR *) messageList,
      sizeof (messageList),
      &numMessages);
printf ("%d message%s%s\n", numMessages,
       (numMessages == 0) ? "s" :
       ((numMessages == 1) ? " : " : "s : "),
       (numMessages > 0) ? messageList : "");
if (rc == SQL_SUCCESS) {
       printf ("Validate succeeded.\n");
}
else {
    driverError (driverHandle, hmod);
}
```

LoadTableFromFile and LoadTableFromFileW

Syntax

```
SQLReturn
LoadTableFromFile  (HDBC      hdbc,
                    SQLCHAR*   TableName,
                    SQLCHAR*   FileName,
                    SQLLEN     ErrorTolerance,
                    SQLLEN     WarningTolerance,
                    SQLCHAR*   ConfigFile,
                    SQLCHAR*   LogFile,
                    SQLCHAR*   DiscardFile,
                    SQLULEN    LoadStart,
                    SQLULEN    LoadCount,
                    SQLULEN    ReadBufferSize)
LoadTableFromFileW  (HDBC      hdbc,
                    SQLWCHAR*  TableName,
                    SQLWCHAR*  FileName,
                    SQLLEN     ErrorTolerance,
```

```
SQLLEN     WarningTolerance,
SQLWCHAR*  ConfigFile,
SQLWCHAR*  LogFile,
SQLWCHAR*  DiscardFile,
SQLULEN    LoadStart,
SQLULEN    LoadCount,
SQLULEN    ReadBufferSize)
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, and SQL_ERROR.

Purpose

LoadTableFromFile (ANSI application) and LoadTablefromFileW (Unicode application) bulk load data from a file to a table. The load operation can create a log file and can also create a discard file that contains rows rejected during the load. The discard file is in the same format as the bulk load data file. After fixing reported issues in the discard file, the bulk load can be reissued using the discard file as the bulk load data file.

The load operation can be configured such that if any errors or warnings occur:

- The operation always completes
- The operation always terminates
- The operation terminates after a certain threshold of warnings or errors is exceeded.

If a load fails, the *LoadStart* and *LoadCount* parameters can be used to control which rows are loaded when a load is restarted after a failure.

Parameters

hdbc

is the driver's connection handle, which is not the handle returned by SQLAllocHandle or SQLAllocConnect. To obtain the driver's connection handle, the application must then use the standard ODBC function SQLGetInfo (*ODBC Conn Handle, SQL_DRIVER_HDBC*).

TableName

is a null-terminated character string that specifies the name of the target database table into which the data is to be loaded. For the Salesforce driver, the value of this parameter can vary. See "Using the TableName parameter with the Salesforce driver" for more information.

FileName

is a null-terminated string that specifies the path (relative or absolute) and file name of the bulk data file from which the data is to be loaded. The file name must be the fully qualified path to the bulk data file.

ErrorTolerance

specifies the number of errors to tolerate before an operation terminates. A value of 0 indicates that no errors are tolerated; the operation fails when the first error is encountered. The default of -1 means that an infinite number of errors is tolerated.

WarningTolerance

specifies the number of warnings to tolerate before an operation terminates. A value of 0 indicates that no warnings are tolerated; the operation fails when the first warning is encountered.The default of -1 means that an infinite number of warnings is tolerated.

ConfigFile

is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk configuration file.

LogFile

is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk log file. The file name must be the fully qualified path to the log file. Events logged to this file are:

- Total number of rows read
- Message for each row that failed to load.
- Total number of rows that failed to load
- Total number of rows successfully loaded

Information about the load is written to this file, preceded by a header. Information about the next load is appended to the end of the file.

If LogFile is NULL, no log file is created.

DiscardFile is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk discard file. The file name must be the fully qualified path to the discard file. Any row that cannot be inserted into database as result of bulk load is added to this file, with the last row to be rejected added to the end of the file.

Information about the load is written to this file, preceded by a header. Information about the next load is appended to the end of the file.

If DiscardFile is NULL, no discard file is created.

LoadStart specifies the first row to be loaded from the data file. Rows are numbered starting with 1. For example, when LoadStart=10, the first 9 rows of the file are skipped and the first row loaded is row 10. This parameter can be used to restart a load after a failure.

LoadCount specifies the number of rows to be loaded from the data file. The bulk load operation loads rows up to the value of LoadCount from the file to the database. It is valid for LoadCount to specify more rows than exist in the data file. The bulk load operation completes successfully when either the LoadCount value has been loaded or the end of the data file is reached. This parameter can be used in conjunction with *LoadStart* to restart a load after a failure.

ReadBufferSize specifies the size, in KB, of the buffer that is used to read the bulk data file for a bulk load operation. The default is 2048.

Example

```
HDBC      hdbc;
HENV      henv;
void      *driverHandle;
HMODULE   hmod;
PLoadTableFromFile loadTableFromFile;
char      tableName[128];
char      fileName[512];
char      configFile[512];
char      logFile[512];
char      discardFile[512];
int       errorTolerance;
int       warningTolerance;
int       loadStart;
int       loadCount;
int       readBufferSize;

/* Get the driver's connection handle from the DM. This handle must be used when calling
 directly into the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HDBC, &driverHandle, 0, NULL);
if (rc != SQL_SUCCESS) {
  ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
  EnvClose (henv, hdbc);
  exit (255);
}
```

```
/* Get the DM's shared library or DLL handle to the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HLIB, &hmod, 0, NULL);
if (rc != SQL_SUCCESS) {
  ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
  EnvClose (henv, hdbc);
  exit (255);
}

loadTableFromFile = (PLoadTableFromFile)
  resolveName (hmod, "LoadTableFromFile");
if (! loadTableFromFile) {
  printf ("Cannot find LoadTableFromFile!\n");
  exit (255);
}
rc = (*loadTableFromFile) (
     driverHandle,
     (const SQLCHAR *) tableName,
     (const SQLCHAR *) fileName,
     errorTolerance, warningTolerance,
     (const SQLCHAR *) configFile,
     (const SQLCHAR *) logFile,
     (const SQLCHAR *) discardFile,
     loadStart, loadCount,
     readBufferSize);
if (rc == SQL_SUCCESS) {
     printf ("Load succeeded.\n");
}
else {
     driverError (driverHandle, hmod);
}
```

See also

Using the TableName parameter with the Salesforce driver on page 108

Using the TableName parameter with the Salesforce driver

The value required in the TableName parameter varies, depending on the bulk operation specified in the SetBulkOperation function. The following paragraphs describe the TableName value based on whether the Bulk Operation type is set to INSERT, DELETE, or UPSERT.

BULK_OPERATION_INSERT

table_name [(*column_list*)]

where:

column_list

is (*columnSpec*[, *columnSpec*]…)

columnSpec

can be *columnName* or `foreignKeyColumnName EXT_ID externalIdColumnName`

The column names define the mapping between columns in the table and columns in the bulk data file. The column names can also indicate which columns are External ID columns.

The SQL equivalent of this function is:

`INSERT INTO` *table_name* `[(` *column_list* `)] VALUES (? … ?)`

BULK_OPERATION_DELETE

table_name (*column_list*)

where:

column_list

is the ID column, which identifies the row to delete.

For DELETE, the ID column is the only valid column in the column list.

The SQL equivalent of this function is:

```
DELETE FROM table_name WHERE <column> = ? AND <column> = ? …
```

BULK_OPERATION_UPDATE

table_name (*column_list*)

where:

column_list

is *ID_column,* <update column>[,<update column>]…

ID_column

must be one of the columns in the column list. The ID column identifies which row to update; the other columns are the list of columns to be updated.

The SQL equivalent of this function is:

```
UPDATE table_name SET <update column> = ? … WHERE <ID column> = ? …
```

BULK_OPERATION_UPSERT

table_name (*column_list*)

where:

column_list

is the same as for INSERT except that at least one of the columns must be identified as an external ID.

For UPSERT, *column_list* can be (*columnSpec*[, *columnSpec*]…)

columnSpec

can be one of the following:

- *columnName*
- foreignKeyColumnName EXT_ID externalIdColumnName
- *extIdColumn* EXT_ID

  where *extIdColumn* is the column that is checked to determine whether the row already exists in the database.

The SQL equivalent of this function is one of the following:

- If no row matching the table's key columns is found:

  ```
  INSERT INTO table_name [(column_list)] VALUES (? … ?)
  ```

- If a row matching the table's key columns is found:

  ```
  UPDATE table_name SET <table column> = ? … WHERE <key column> = ? …
  ```

SetBulkOperation (Salesforce driver only)

Syntax

```
SQLReturn
SetBulkOperation  (HDBC      hdbc,
                   SQLULEN   Operation)
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, SQL_NO_DATA, and SQL_ERROR.

Purpose

Specifies the bulk operation to be performed when either the LoadTableFromFile and LoadTableFromFileW method is called. The bulk operation remains set until SetBulkOperation is called again. When a connection is established, the initial bulk operation is BULK_OPERATION_INSERT.

Parameters

hdbc

is the driver's connection handle, which is not the handle returned by SQLAllocHandle or SQLAllocConnect. To obtain the driver's connection handle, the application must use SQLGetInfo (*ODBC Conn Handle,SQL_DRIVER_HDBC)*.

Operation

is an integer value that specifies the bulk operation to set on the connection. It can have one of the following values:

- 1 - BULK_OPERATION_INSERT
- 2 - BULK_OPERATION_UPDATE
- 3- BULK_OPERATION_DELETE
- 4 - BULK_OPERATION_UPSERT

Example

```
HDBC      hdbc;
HENV      henv;
void      *driverHandle;
HMODULE   hmod;
PSetBulkOperation setBulkOperation;
/* Get the driver's connection handle from the DM. This handle must be used when calling
 directly into the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HDBC, &driverHandle, 0, NULL);
if (rc != SQL_SUCCESS) {
   ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
```

```
   EnvClose (henv, hdbc);
   exit (255);
}
/* Get the DM's shared library or DLL handle to the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HLIB, &hmod, 0, NULL);
if (rc != SQL_SUCCESS) {
   ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
   EnvClose (henv, hdbc);
   exit (255);
}
/* Set the Bulk Operation type to DELETE. Any subsequent call to LoadTableFromFile(W)
will result in a bulk delete of the rows specified. */

setBulkOperation = (PSetBulkOperation)
   resolveName (hmod, "SetBulkOperation");
if (! setBulkOperation) {
   printf ("Cannot find SetBulkOperation!\n");
   exit (255);
}

rc = (*setBulkOperation) (
      driverHandle,
      BULK_OPERATION_DELETE);
if (rc == SQL_SUCCESS) {
    printf ("Set Bulk operation(DELETE) succeeded.\n");
}else {
     driverError (driverHandle, hmod);
}
/*  */
```

See also

LoadTableFromFile and LoadTableFromFileW on page 105

GetBulkOperation (Salesforce driver only)

Syntax

```
SQLReturn
GetBulkOperation  (HDBC        hdbc,
                   SQLULEN    Operation)
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, and SQL_ERROR.

Purpose

Returns the bulk operation currently set on the connection. The bulk operation specifies the operation to be performed when either the LoadTableFromFile or LoadTableFromFileW method is called.

Parameters

hdbc

is the driver's connection handle, which is not the handle returned by SQLAllocHandle or SQLAllocConnect. To obtain the driver's connection handle, the application must then use the standard ODBC function SQLGetInfo (ODBC Conn Handle, SQL_DRIVER_HDBC).

Operation

is a pointer to the location where current bulk operation specified for the connection is returned. The returned value is one of the operation values defined by SetBulkOperation.

Example

```
HDBC      hdbc;
HENV      henv;
void      *driverHandle;
HMODULE  hmod;
PGetBulkOperation getBulkOperation;
SQLULEN bulkOperationType;

/* Get the driver's connection handle from the DM. This handle must be used when calling
 directly into the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HDBC, &driverHandle, 0, NULL);
if (rc != SQL_SUCCESS) {
   ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
   EnvClose (henv, hdbc);
   exit (255);
}

/* Get the DM's shared library or DLL handle to the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HLIB, &hmod, 0, NULL);
if (rc != SQL_SUCCESS) {
   ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
   EnvClose (henv, hdbc);
   exit (255);
}

/* Get the current value for bulk operation. */

getBulkOperation = (PGetBulkOperation)
resolveName (hmod, "GetBulkOperation");
if (! getBulkOperation) {
   printf ("Cannot find GetBulkOperation!\n");
   exit (255);
}

rc = (*getBulkOperation) (
driverHandle,
&bulkOperationType);
if (rc == SQL_SUCCESS) {
    printf ("Current bulk operation is: %u.\n", bulkOperationType);
}
else {
    driverError (driverHandle, hmod);
}

/*  */
```

See also

LoadTableFromFile and LoadTableFromFileW on page 105
SetBulkOperation (Salesforce driver only) on page 110

DataDirect Bulk Load statement attributes

In addition to exporting tables with the ExportTableToFile methods, result sets can be exported to a bulk load data file through the use of two DataDirect statement attributes, SQL_BULK_EXPORT_PARAMS and SQL_BULK_EXPORT. SQL_BULK_EXPORT_PARAMS is used to configure information about where and how the data is to be exported. SQL_BULK_EXPORT begins the bulk export operation.

SQL_BULK_EXPORT_PARAMS

The ValuePtr argument to SQLSetStmtAttr or SQLSetStmtAttrW when the attribute argument is SQL_BULK_EXPORT_PARAMS is a pointer to a BulkExportParams structure. The definitions of the fields in the BulkExportParams structure are the same as the corresponding arguments in the ExportTableToFile and ExportTableToFileW methods except that the generation of the log file is controlled by the EnableLogging field. When EnableLogging is set to 1, the driver writes events that occur during the export to a log file. Events logged to this file are:

- A message for each row that failed to export.
- Total number of rows fetched
- Total number of rows successfully exported
- Total number of rows that failed to export

The log file is located in the same directory as the bulk load data file and has the same base name as the bulk load data file with a .log extension. When EnableLogging is set to 0, no logging takes place

If the bulk export parameters are not set prior to setting the SQL_BULK_EXPORT attribute, the driver uses the current driver code page value, defaults EnableLogging to 1 (enabled), and defaults ErrorTolerance and WarningTolerance to -1 (infinite).

The SQL_BULK_EXPORT_PARAMS structure is as follows:

```
struct BulkExportParams {
  SQLLEN  Version;               /* Must be the value 1 */
  SQLLEN  IANAAppCodePage;
  SQLLEN  EnableLogging;
  SQLLEN  ErrorTolerance;
  SQLLEN  WarningTolerance;
};
```

SQL_BULK_EXPORT

The ValuePtr argument to SQLSetStmtAttr or SQLSetStmtAttrW when the attribute argument is SQL_BULK_EXPORT is a pointer to a string that specifies the file name of the bulk load data file to which the data in the result set will be exported.

Result set export occurs when the SQL_BULK_EXPORT statement attribute is set. If using the SQL_BULK_EXPORT_PARAMS attribute to set values for the bulk export parameters, the SQL_BULK_EXPORT_PARAMS attribute must be set prior to setting the SQL_BULK_EXPORT attribute. Once set, the bulk export parameters remain set for the life of the statement. If the bulk export parameters are not set prior to setting the SQL_BULK_EXPORT attribute, the driver uses the current driver code page value, defaults EnableLogging to 1 (enabled), and defaults ErrorTolerance and WarningTolerance to -1 (infinite).

Both a bulk load data file and a bulk load configuration file are produced by this operation. The configuration file has the same base name as the bulk load data file, but with an XML extension. The configuration file is created in the same directory as the bulk load data file.

13

DataDirect connection pooling

Connection pooling allows you to *reuse* connections rather than creating a new one every time the driver needs to establish a connection to the underlying database. Your Progress DataDirect driver enables connection pooling without requiring changes to your client application.

Note: Connection pooling works only with connections that are established using SQLConnect or SQLDriverConnect with the SQL_DRIVER_NO_PROMPT argument and only with applications that are thread-enabled.

DataDirect connection pooling that is implemented by the DataDirect driver is different than connection pooling implemented by the Windows Driver Manager. The Windows Driver Manager opens connections dynamically, up to the limits of memory and server resources. DataDirect connection pooling, however, allows you to control the number of connections in a pool through the Min Pool Size (minimum number of connections in a pool) and Max Pool Size (maximum number of connections in a pool) connection options. In addition, DataDirect connection pooling is cross-platform, allowing it to operate on UNIX and Linux.

Important: On a Windows system, do not use both Windows Driver Manager connection pooling and DataDirect connection pooling at the same time.

The following topics provide general information on how DataDirect connection pooling works. For detailed information on connection pooling options, refer to "Connection option descriptions" in the user's guide for your driver.

For details, see the following topics:

- Creating a connection pool
- Adding connections to a pool

Creating a connection pool

Each connection pool is associated with a specific connection string. By default, the connection pool is created when the first connection with a unique connection string connects to the data source. The pool is populated with connections up to the minimum pool size before the first connection is returned. Additional connections can be added until the pool reaches the maximum pool size. If the Max Pool Size option is set to `10` and all connections are active, a request for an eleventh connection has to wait in queue for one of the 10 pool connections to become idle. The pool remains active until the process ends or the driver is unloaded.

If a new connection is opened and the connection string does not exactly match an existing pool, a new pool must be created. By using the same connection string, you can enhance the performance and scalability of your application.

Adding connections to a pool

A connection pool is created in the process of creating each unique connection string that an application uses. When a pool is created, it is populated with enough connections to satisfy the minimum pool size requirement, set by the Min Pool Size connection option. The maximum pool size is set by the Max Pool Size connection option. If an application needs more connections than the number set by Min Pool Size, the driver allocates additional connections to the pool until the number of connections reaches the value set by Max Pool Size.

Once the maximum pool size has been reached and no usable connection is available to satisfy a connection request, the request is queued in the driver. The driver waits for the length of time specified in the Login Timeout connection option for a usable connection to return to the application. If this time period expires and a connection has not become available, the driver returns an error to the application.

A connection is returned to the pool when the application calls SQLDisconnect. Your application is still responsible for freeing the handle, but this does not result in the database session ending.

Removing connections from a pool

A connection is removed from a connection pool when it exceeds its lifetime as determined by the Load Balance Timeout connection option. In addition, DataDirect has created connection attributes described in the following table to give your application the ability to reset connection pools. If connections are in use at the time of these calls, they are marked appropriately. When SQLDisconnect is called, the connections are discarded instead of being returned to the pool.

Table 15: Pool Reset Connection Attributes

Connection Attribute	Description
SQL_ATTR_CLEAR_POOLS Value: SQL_CLEAR_ALL_CONN_POOL	Calling SQLSetConnectAttr (SQL_ATTR_CLEAR_POOLS, SQL_CLEAR_ALL_CONN_POOL) clears all the connection pools associated with the driver that created the connection. This is a write-only connection attribute. The driver returns an error if SQLGetConnectAttr (SQL_ATTR_CLEAR_POOLS) is called.
SQL_ATTR_CLEAR_POOLS Value: SQL_CLEAR_CURRENT_CONN_POOL	Calling SQLSetConnectAttr (SQL_ATTR_CLEAR_POOLS, SQL_CLEAR_CURRENT_CONN_POOL) clears the connection pool that is associated with the current connection. This is a write-only connection attribute. The driver returns an error if SQLGetConnectAttr (SQL_ATTR_CLEAR_POOLS) is called.

Note: By default, if removing a connection causes the number of connections to drop below the number specified in the Min Pool Size option, a new connection is not created until an application needs one.

Handling dead connections in a pool

What happens when an idle connection loses its physical connection to the database? For example, suppose the database server is rebooted or the network experiences a temporary interruption. An application that attempts to connect could receive errors because the physical connection to the database has been lost.

Your Progress DataDirect *for* ODBC driver handles this situation transparently to the user. The application does not receive any errors on the connection attempt because the driver simply returns a connection from a connection pool. The first time the connection handle is used to execute a SQL statement, the driver detects that the physical connection to the server has been lost and attempts to reconnect to the server *before* executing the SQL statement. If the driver can reconnect to the server, the result of the SQL execution is returned to the application; no errors are returned to the application.

The driver uses connection failover option values, if they are enabled, when attempting this seamless reconnection; however, it attempts to reconnect even if these options are not enabled.

Note: If the driver cannot reconnect to the server (for example, because the server is still down), an error is returned indicating that the reconnect attempt failed, along with specifics about the reason the connection failed.

The technique that Progress DataDirect uses for handling dead connections in connection pools allows for maximum performance of the connection pooling mechanism. Some drivers periodically test the server with a dummy SQL statement while the connections sit idle. Other drivers test the server when the application requests the use of the connection from the connection pool. Both of these approaches add round trips to the database server and ultimately slow down the application during normal operation.

Connection pool statistics

Progress DataDirect has created a connection attribute to monitor the status of the DataDirect *for* ODBC connection pools. This attribute, which is described in the following table, allows your application to fetch statistics for the pool to which a connection belongs.

Table 16: Pool Statistics Connection Attribute

Connection Attribute	Description
SQL_ATTR_POOL_INFO Value: SQL_GET_POOL_INFO	Calling SQLGetConnectAttr (SQL_ATTR_POOL_INF, SQL_GET_POOL_INFO) returns a PoolInfoStruct that contains the statistics for the connection pool to which this connection belongs. This PoolInfoStruct is defined in `qesqlext.h`. For example:`SQLGetConnectAttr(hdbc, SQL_ATTR_POOL_INFO, PoolInfoStruct *, SQL_LEN_BINARY_ATTR(PoolInfoStruct), &len);`This is a read-only connection attribute. The driver returns an error if SQLSetConnectAttr (SQL_ATTR_POOL_INFO) is called.

Summary of pooling-related options

The following table summarizes how connection pooling-related connection options work with the drivers. For more detailed information, refer to "Connection option descriptions" in the user's guide for your driver.

Table 17: Summary: Connection Pooling Connection Options

Option	Characteristic
Connection Pooling	Enables connection pooling.
Connection Reset	Resets a connection that is removed from the connection pool to the initial configuration settings of the connection.
Load Balance Timeout	An integer value to specify the amount of time, in seconds, to keep connections open in a connection pool.
Max Pool Size	An integer value to specify the maximum number of connections within a single pool.
Min Pool Size	An integer value to specify the minimum number of connections that are opened and placed in a connection pool when it is created.

14

Threading

The ODBC specification mandates that all drivers must be thread-safe, that is, drivers must not fail when database requests are made on separate threads. It is a common misperception that issuing requests on separate threads always results in improved throughput. Because of network transport and database server limitations, some drivers serialize threaded requests to the server to ensure thread safety.

The ODBC 3.0 specification does not provide a method to find out how a driver services threaded requests, although this information is useful to an application. All the Progress DataDirect ODBC drivers provide this information to the user through the SQLGetInfo information type 1028.

The result of calling SQLGetInfo with 1028 is a SQL_USMALLINT flag that denotes the session's thread model. A return value of 0 denotes that the session is fully thread-enabled and that all requests use the threaded model. A return value of 1 denotes that the session is restricted at the connection level. Sessions of this type are fully thread-enabled when simultaneous threaded requests are made with statement handles that do not share the same connection handle. In this model, if multiple requests are made from the same connection, the first request received by the driver is processed immediately and all subsequent requests are serialized. A return value of 2 denotes that the session is thread-impaired and all requests are serialized by the driver.

Consider the following code fragment:

```
rc = SQLGetInfo (hdbc, 1028, &ThreadModel, NULL, NULL);

If (rc == SQL_SUCCESS) {
   // driver is a DataDirect driver that can report threading information

   if (ThreadModel == 0)
   // driver is unconditionally thread-enabled; application can take advantage of
   // threading

   else if (ThreadModel == 1)
   // driver is thread-enabled when thread requests are from different connections
   // some applications can take advantage of threading

   else if (ThreadModel == 2)
   // driver is thread-impaired; application should only use threads if it reduces
   // program complexity
```

```
}
else
   // driver is not guaranteed to be thread-safe; use threading at your own risk
```

15

WorkAround options

Progress DataDirect has included non-standard connection options (workarounds) that enable you to take full advantage of packaged ODBC-enabled applications requiring non-standard or extended behavior. When using workaround options, a separate user data source should be created for each application. We recommend that you consult with Progress Technical Support for assistance.

On Windows, you can use the Extended Options field on the Advanced tab of the driver's Setup dialog box to set workaround options for most drivers. However, some drivers do not have the Extended Options field.

Alternatively, workaround options can be configured in the following way.

After you create the data source:

- On Windows, using the registry editor REGEDIT, open the HKEY_CURRENT_USER\SOFTWARE\ODBC\ODBC.INI section of the registry. Select the data source that you created.
- On UNIX/Linux/macOS, using a text editor, open the odbc.ini file to edit the data source that you created.

Add the string WorkArounds= (or WorkArounds2=) with a value of n (WorkArounds=n or WorkArounds2=n), where the value *n* is the cumulative value of all options added together. For example, if you wanted to use both WorkArounds=1 and WorkArounds=8, you would enter in the data source:

```
WorkArounds=9
```

Warning: Each of these options has potential side effects related to its use. An option should only be used to address the specific problem for which it was designed. For example, WorkArounds=2 causes the driver to report that database qualifiers are not supported, even when they are. As a result, applications that use qualifiers may not perform correctly when this option is enabled.

The following list includes both WorkArounds and WorkArounds2.

WorkArounds=1. Enabling this option causes the driver to return 1 instead of 0 if the value for SQL_CURSOR_COMMIT_BEHAVIOR or SQL_CURSOR_ROLLBACK_BEHAVIOR is 0. Statements are prepared again by the driver.

WorkArounds=2. Enabling this option causes the driver to report that database qualifiers are not supported. Some applications cannot process database qualifiers.

WorkArounds=8. Enabling this option causes the driver to return 1 instead of -1 for SQLRowCount. If an ODBC driver cannot determine the number of rows affected by an Insert, Update, or Delete statement, it may return -1 in SQLRowCount. This may cause an error in some products.

WorkArounds=16. Enabling this option causes the driver not to return an INDEX_QUALIFIER. For SQLStatistics, if an ODBC driver reports an INDEX_QUALIFIER that contains a period, some applications return a "tablename is not a valid name" error.

WorkArounds=32. Enabling this option causes the driver to re-bind columns after calling SQLExecute for prepared statements.

WorkArounds=64. Enabling this option results in a column name of C*position* where *position* is the ordinal position in the result set. For example, "SELECT col1, col2+col3 FROM table1" produces the column names "col1" and C2. For result columns that are expressions, SQLColAttributes/SQL_COLUMN_NAME returns an empty string. Use this option for applications that cannot process empty string column names.

WorkArounds=256. Enabling this option causes the value of SQLGetInfo/SQL_ACTIVE_CONNECTIONS to be returned as 1.

WorkArounds=512. Enabling this option prevents ROWID results. This option forces the SQLSpecialColumns function to return a unique index as returned from SQLStatistics.

WorkArounds=2048. Enabling this option causes DATABASE= instead of DB= to be returned. For some data sources, Microsoft Access performs more efficiently when the output connection string of SQLDriverConnect returns DATABASE= instead of DB=.

WorkArounds=65536. Enabling this option strips trailing zeros from decimal results, which prevents Microsoft Access from issuing an error when decimal columns containing trailing zeros are included in the unique index.

WorkArounds=131072. Enabling this option turns all occurrences of the double quote character (") into the accent grave character (`). Some applications always quote identifiers with double quotes. Double quoting can cause problems for data sources that do not return SQLGetInfo/SQL_IDENTIFIER_QUOTE_CHAR = *double_quote*.

WorkArounds=524288. Enabling this option forces the maximum precision/scale settings. The Microsoft Foundation Classes (MFC) bind all SQL_DECIMAL parameters with a fixed precision and scale, which can cause truncation errors.

WorkArounds=1048576. Enabling this option overrides the specified precision and sets the precision to 256. Some applications incorrectly specify a precision of 0 for character types when the value will be SQL_NULL_DATA.

WorkArounds=2097152. Enabling this option overrides the specified precision and sets the precision to 2000. Some applications incorrectly specify a precision of -1 for character types.

WorkArounds=4194304. Enabling this option converts, for PowerBuilder users, all catalog function arguments to uppercase unless they are quoted.

WorkArounds=16777216. Enabling this option allows MS Access to retrieve Unicode data types as it expects the default conversion to be to SQL_C_CHAR and not SQL_C_WCHAR.

WorkArounds=33554432. Enabling this option prevents MS Access from failing when SQLError returns an extremely long error message.

WorkArounds=67108864. Enabling this option allows parameter bindings to work correctly with MSDASQL.

WorkArounds=536870912. Enabling this option allows re-binding of parameters after calling SQLExecute for prepared statements.

WorkArounds=1073741824. Enabling this option addresses the assumption by the application that ORDER BY columns do not have to be in the SELECT list. This assumption may be incorrect for data sources such as Informix.

WorkArounds2=2. Enabling this option causes the driver to ignore the ColumnSize/DecimalDigits specified by the application and use the database defaults instead. Some applications incorrectly specify the ColumnSize/DecimalDigits when binding timestamp parameters.

WorkArounds2=4. Enabling this option reverses the order in which Microsoft Access returns native types so that Access uses the most appropriate native type. Microsoft Access uses the last native type mapping, as returned by SQLGetTypeInfo, for a given SQL type.

WorkArounds2=8. Enabling this option causes the driver to add the bindoffset in the ARD to the pointers returned by SQLParamData. This is to work around an MSDASQL problem.

WorkArounds2=16. Enabling this option causes the driver to ignore calls to SQLFreeStmt(RESET_PARAMS) and only return success without taking other action. It also causes parameter validation not to use the bind offset when validating the charoctetlength buffer. This is to work around a MSDASQL problem.

WorkArounds2=24. Enabling this option allows a flat-file driver, such as dBASE, to operate properly under MSDASQL.

WorkArounds2=32. Enabling this option appends "DSN=" to a connection string if it is not already included. Microsoft Access requires "DSN" to be included in a connection string.

WorkArounds2=128. Enabling this option causes 0 to be returned by SQLGetInfo(SQL_ACTIVE_STATEMENTS). Some applications open extra connections if SQLGetInfo(SQL_ACTIVE_STATEMENTS) does not return 0.

WorkArounds2=256. Enabling this option causes the driver to return Buffer Size for Long Data on calls to SQLGetData with a buffer size of 0 on columns of SQL type SQL_LONGVARCHAR or SQL_LONGVARBINARY. Applications should always set this workaround when using MSDASQL and retrieving long data.

WorkArounds2=512. Enabling this option causes the flat-file drivers to return old literal prefixes and suffixes for date, time, and timestamp data types. Microsoft Query 2000 does not correctly handle the ODBC escapes that are currently returned as literal prefix and literal suffix.

WorkArounds2=1024. Enabling this option causes the driver to return "N" for SQLGetInfo(SQL_MULT_RESULT_SETS). ADO incorrectly interprets SQLGetInfo(SQL_MULT_RESULT_SETS) to mean that the contents of the last result set returned from a stored procedure are the output parameters for the stored procedure.

WorkArounds2=2048. Enabling this option causes the driver to accept 2.x SQL type defines as valid. ODBC 3.x applications that use the ODBC cursor library receive errors on bindings for SQL_DATE, SQL_TIME, and SQL_TIMESTAMP columns. The cursor library incorrectly rebinds these columns with the ODBC 2.x type defines.

WorkArounds2=4096. Enabling this option causes the driver to internally adjust the length of empty strings. The ODBC Driver Manager incorrectly translates lengths of empty strings when a Unicode-enabled application uses a non-Unicode driver. Use this workaround only if your application is Unicode-enabled.

WorkArounds2=8192. Enabling this option causes Microsoft Access not to pass the error -7748. Microsoft Access only asks for data as a two-byte SQL_C_WCHAR, which is an insufficient buffer size to store the UCS2 character and the null terminator; thus, the driver returns a warning, "01004 Data truncated" and returns a null character to Microsoft Access. Microsoft Access then passes error -7748.